AFIT/GCE/ENG/93D-07

PARTITIONING STRUCTURAL VHDL CIRCUITS FOR PARALLEL EXECUTION ON HYPERCUBES

THESIS

Presented to the Faculty of the Graduate School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Computer Engineering

Kevin L. Kapp, B.S.E.E.

Capt, USAF

Dec, 1993

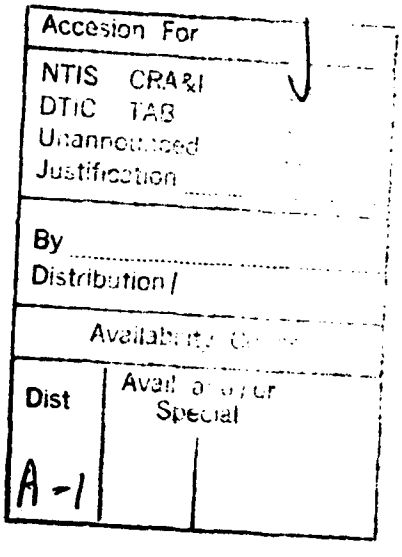
AFIT/GCE/ENG/93D-07

PARTITIONING STRUCTURAL VHDL CIRCUITS FOR PARALLEL EXECUTION ON HYPERCUBES

THESIS

Kevin L. Kapp, B.S.E.E.
Capt, USAF

AFIT/GCE/ENG/93D-07



Acknowledgments

I would like to offer my sincere thanks and appreciation to my thesis committee for their guidance and support during this research effort. Although I was given considerable independence to pursue my own ideas, it was only with their guidance that this was possible.

- Dr. Hartrum expended considerable effort teaching the "ins and outs" of parallel discrete event simulation and synchronization protocols. He has tremendous patience, always willing to repeat a difficult concept until it sinks in. He understands the importance of building a good foundation upon which more advanced research can be built.
- Maj Christensen's enthusiastic support and assistance was invaluable during the early stages of my research. He built the original graph partitioning tool to perform random partitions. He expended considerable effort debugging the associative memory array VHDL source code to ensure it was compatible with VSIM. He made several improvements and corrections to VSIM and its associated utility programs.
- LtCol Wailes gave me the freedom to define my own partitioning approach and research methodology. By keeping his eye on the "big picture," he made sure I stayed focused on my objectives and did not become mired in side issues. He kept me on track and helped me to keep my successes and failures in perspective.

Perhaps the most important thank you goes to my wife, Diane. Without her continuous support, patience, and understanding, this thesis would not have been completed. She gave me the time I needed to meet the demands of graduate-level

academics, but made sure I also took time out for my family and myself. She was always there when I needed her.

Finally, I would like to thank my son Joseph A. Kapp. Although only 1 1/2, he has forced me to put my thesis, my career, and my life in proper perspective. Watching him grow up this past eighteen months has changed my life in innumerable ways.

Kevin L. Kapp

Table of Contents

List of Figures

List of Tables

Abstract

Distributing simulations among multiple processors is one approach to reducing VHDL simulation time for large VLSI circuit designs. However, parallel simulation introduces the problem of how to partition the logic gates and system behaviors among the available processors in order to obtain maximum speedup. This research investigates deliberate partitioning algorithms that account for the complex inter-dependency structure of the circuit behaviors. Once an initial partition has been obtained, a *border annealing* algorithm is used to iteratively improve the partition. In addition, methods of measuring the cost of a partition and relating it to the resulting simulation performance are investigated. Structural circuits ranging from one thousand to over four thousand behaviors are simulated. The deliberate partitions consistently provided superior speedup to a random distribution of the circuit behaviors.

PARTITIONING STRUCTURAL VHDL CIRCUITS FOR PARALLEL EXECUTION ON HYPERCUBES

I. Introduction

1.1 Background

Modern integrated circuit designs are rapidly growing larger and more complex, with chip transistor counts increasing by approximately 25% per year, doubling every three years (13:17). In order to reduce chip costs and turnaround times, designers use sophisticated simulation tools to validate their designs prior to chip fabrication (4:1). The Department of Defense (DOD) established the Very High Speed Integrated Circuit (VHSIC) program in 1979 with the primary objective of advancing the state of the art in the areas of large scale circuit design and manufacturing technology (15:1). As part of this program, the VHSIC Hardware Description Language (VHDL) program began in 1981 with the goal of developing a standard simulation language for the support of hardware design (15, 4).

As the size and complexity of circuit designs continue their upward trend, there is a growing need to increase the speed of the VHDL simulations. Slow sequential simulations result in a longer iterative design process and increase the cost of the final product. In an effort to achieve the desired performance improvement, the Advanced Research Projects Agency (ARPA) has sponsored the QUEST project with the objective of obtaining a thousand-fold speedup in large VHDL simulations using the commercial Intermetrics VHDL simulator running on a VAX 8650 as the baseline (6:2, 19:1-1).

Previous AFIT research has investigated the possibility of achieving speedup by distributing the VHDL simulations over multiple processors for parallel execution. By effectively sharing the simulation workload among multiple processors, simulations of complex chip designs can be run faster, resulting in a more efficient and cost-effective design cycle.

AFIT research in 1991-92 focused on the internal data structures used in the Intermetrics commercial VHDL simulator which runs in a sequential mode. A method of intercepting the intermediate C source code from the Intermetrics compiler, transforming it into parallel models, and executing the transformed code in parallel with correct results on Intel iPSC/2 and iPSC/860 hypercubes has been demonstrated (8, 4). The result of this research is a parallel VHDL simulator, referred to as VSIM, that implements a selected subset of the standard VHDL language (4).

Breeden's results demonstrated that speedup on multiple processors can be achieved under limited circumstances using a random partitioning of the VHDL behaviors[1] among the processors of the hypercube (4). In his random partitioning approach, the objective was to randomly assign an equal number of behaviors to each processor without considering their complex inter-dependency relationships. As a result, the speedup results were significantly less than optimal, and fell off rapidly as the number of processors was increased due to increases in communications overhead. This thesis research effort focuses on the development of efficient and effective partitioning strategies to map the logical VHDL behaviors to the physical processors of a hypercube in order to take maximum advantage of the parallelism available in the simulation application.

---

[1] A *behavior* is an executable VHDL process representing a logic gate, source signal, or other simple VHDL process.

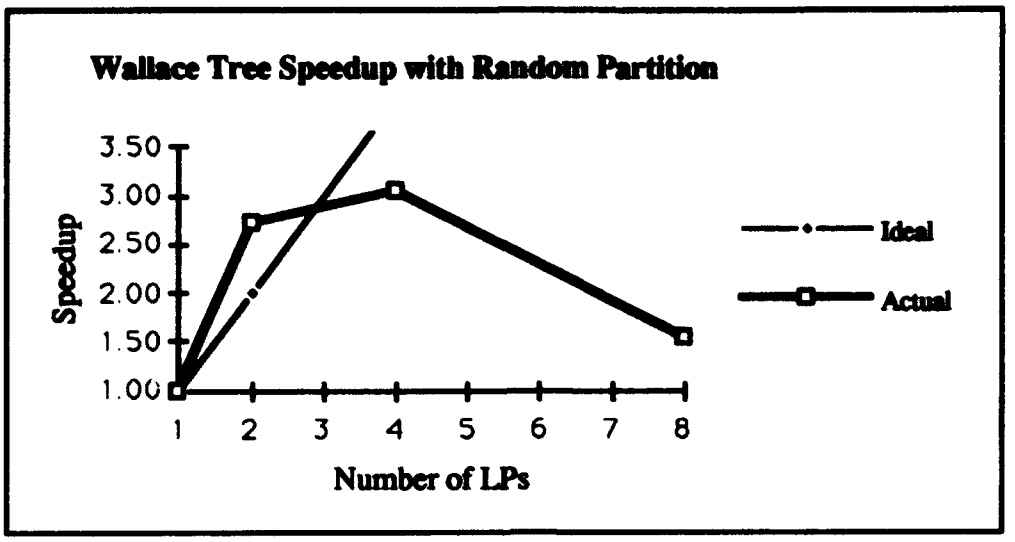


Figure 1. Speedup Curve for Wallace Tree with Random Partitioning

1.2 Problem Statement

AFIT's parallel VHDL simulator, VSIM, has been validated on circuits as large as an 8 x 8 Wallace Tree Multiplier, containing over 1000 VHDL behaviors, on both an 8-node iPSC/2 and an 8-node iPSC/860 hypercube (4). However, in order to maximize the benefits of parallelization, a deliberate partitioning strategy is required that takes into account the complex inter-dependency relationships of the VHDL behaviors when mapping them onto the physical processors of the parallel system. Otherwise, the communications overhead required to maintain synchronization among the processors will negate the potential speedup benefits. For example, Figure 1 shows how the speedup curve for the Wallace Tree Multiplier on the iPSC/2 with a random partitioning of the behaviors takes a downward turn as the number of processors is increased past four.

1.3 Research Objectives

The primary objective of this thesis is to demonstrate improved speedup over random partitioning in the simulation of medium to large sized VHDL circuits using the VSIM

parallel simulator. This will be accomplished through the use of a deliberate partitioning strategy. Specific research goals include:

- Developing an efficient and effective partitioning strategy that accounts for the complex inter-dependency structure of the VHDL circuit being simulated.
- Investigating methods of computing the *cost* of a partition.
- Quantifying the relationship between the cost of a partition and the resulting performance of the simulation.
- Demonstrating improved speedup over a random partitioning using a variety of VHDL circuits.

1.4 Assumptions

The research by Comeau provided the foundation for the transformation of Intermetrics VHDL models into models that can be executed in a parallel environment (8). Breeden built upon this work, automating the transformation process, and validating the results of the parallel simulator VSIM (4). Building upon their research, the following assumptions are made in this thesis:

- The subset of the standard VHDL implemented by VSIM, as described in (4), is adequate to demonstrate the feasibility and effectiveness of various partitioning strategies.
- The commercial Intermetrics VHDL compiler, version 2.1, September 1990, will be used to provide the sequential VHDL models (4).
- The conservative Chandy-Misra algorithm for parallel discrete event simulation (PDES) is used to maintain synchronization between the processors of the parallel system. Using the SPECTRUM[2] testbed, the null-message protocol is used to provide deadlock avoidance (4). To maintain consistency with the AFIT

[2] Simulation Protocol Evaluation on a Concurrent Testbed using ReUsable Modules (20).

simulation environment, this synchronization protocol will not be significantly altered for this thesis.

- Secondary storage input/output (I/O) during parallel simulations on the Intel hypercubes has been shown to overwhelm the benefits of parallelization (4:80). This thesis will focus on achieving computational speedup only. It is assumed that other research will effectively address the architectural issues associated with the large I/O requirements of PDES applications.
- Under the SPECTRUM simulation environment, individual VHDL behaviors are grouped into logical processes (LPs) to increase the granularity of the application tasks. The research in this thesis makes the assumption of one LP per physical processor. This assumption eliminates the context switching and message passing overhead encountered when multiplexing several LPs on a single processor.
- A graph-based behavior dependency representation will provide the information necessary to make sound partitioning decisions in an efficient manner.

1.5 Scope

The following list outlines the limits on the scope of this research effort:

- Finding an *optimal* solution to the problem of mapping N inter-dependent tasks onto P processors is known to be NP-Complete (22:142). This research will seek an efficient and effective heuristic approach that results in consistently *good* solutions, though they may be sub-optimal.
- The subset of the standard VHDL supported by VSIM will not be extended as part of this research effort.
- Circuit descriptions used to validate various partitioning strategies will be limited to less than 5000 behaviors. To implement circuits much larger than this limit in a realistic manner will require extensions to the VHDL subset supported by VSIM.

- This research will not alter the conservative null-message parallel discrete event simulation (PDES) protocol currently implemented by VSIM except when such alterations directly support the validation of a partitioning strategy.

1.6 Limitations

The limitations of the VSIM parallel VHDL simulator are described in (4:4-6). No new limitations on VSIM are imposed as a result of this thesis effort. However, the partitioning tool implemented as part of this thesis has been limited to a maximum of 128 LPs due to the memory required for the data structures used.

1.7 Thesis Overview

Chapter 2 reviews several general approaches to solving the problem of efficiently mapping N tasks onto P processors as found in the current literature. Chapter 3 gives the background on the implementation of VSIM and the SPECTRUM simulation environment. This information leads into a discussion of the specific requirements for a parallel VHDL partitioning strategy. Implementation of this strategy is discussed in Chapter 4. Chapter 5 discusses the research methodology and results. Finally, Chapter 6 presents the conclusions formulated during this research and gives recommendations for future research.

1.8 Summary

The need for this research stems from the rapid increase in the size and complexity of modern large-scale integrated circuit designs. Current commercial VHDL simulators execute in a sequential manner, leading to long design cycles for extremely large circuits. One approach to achieving the desired speedup is through distribution of the simulation load among multiple processors in a parallel system. Previous AFIT research has

validated the concept of parallel VHDL simulation through the development of VSIM. This research investigates methods of partitioning the VHDL circuits among the parallel processors in order the maximize the speedup obtained through parallelization.

II. Background

2.1 Overview

This chapter presents a discussion of previous research relating to the parallel program *Mapping Problem* and how those results might be applied to the specific problem of partitioning structural VHDL circuit descriptions for parallel simulation. To facilitate this discussion, several characteristics of an *ideal* partitioning strategy are first presented.

2.2 The VHDL Mapping Problem

2.2.1 The Parallel Programming Mapping Problem. In the context of parallel programming applications, the *mapping problem* is defined as the binding of the logical components of the parallel application program to the physical resources of the target parallel system such that some desired performance criterion is optimized (22:141). For example, it is usually desired to map the application in such a way that the total execution time is minimized. Optimal solutions to the general mapping problem have been shown to be NP-complete and no polynomial time algorithm for their solution is known to exist (17:63, 22:142). As a result, sub-optimal solutions are often pursued using various heuristic methods (22, 17). The logical-to-physical binding of a parallel application controls the utilization of the parallel system and directly affects the amount of time and memory required to complete program execution.

The mapping problem arises when the number of processes (i.e. tasks) required by the parallel application is greater than the number of available processors (*cardinality variation*), or when the task-dependency structure of the parallel application differs from the physical interconnection structure of the parallel system (*topological variation*) (2).

2.2.2 Partitioning VSIM. To date, no effort has been made at developing an optimal (or near-optimal) partitioning strategy for mapping VHDL behaviors to fully exploit the parallelism available in large VHDL simulations using the VSIM simulator (4). In the parallel VHDL simulation environment created by VSIM, cardinality variation can be dealt with by grouping VHDL behaviors into logical processes (LPs) which comprise the concurrent tasks managed by the SPECTRUM testbed. A large VHDL circuit may contain hundreds of thousands of behaviors with a complex interdependency structure. Assuming the assignment of 1 LP per physical processor, there is likely to be hundreds, or even thousands, of behaviors per LP. As a result, the behavior grouping, or partitioning, is likely to be a critical factor in the relative performance of the parallel simulation.

The two key objectives of most strategies that have been proposed for the general parallel program mapping problem are achieving a balanced computation load among all of the processors, and minimizing the inter-processor communication. The former deals with making efficient use of all of the processor resources, while the latter deals with reducing non-productive overheads such as message setup and transfer times.

The general mapping problem can be divided into two sub-problems: *job scheduling* and *task allocation* (21:1408). The goal of job scheduling is to obtain maximum system utilization by scheduling independent jobs among the processors in a distributed system. This involves a dynamic scheduling ability as old jobs are completed and new ones are submitted. In contrast, the task allocation problem involves the allocation of several inter-dependent tasks of a single program among the processors in a distributed or parallel system. The goal of task allocation is to minimize the completion time of the single application program. The task allocation problem has been approached separately as both a dynamic and static allocation problem, with the latter being desirable if the interdependencies of the task structure can be statically defined a priori (21:1409). In the

VSIM environment, each individual VHDL behavior is equivalent to a task.[3] In addition, the inter-dependency structure of the behaviors is known prior to simulation and is static. Therefore, throughout this thesis, discussion of the mapping problem implies the static task allocation problem.

2.3 Characteristics of an Ideal Partitioning Strategy

Before evaluating various heuristic solutions to the general parallel program mapping problem, it is useful to discuss some characteristics of an *ideal* partitioning strategy that can be used for comparison purposes. In the context of this thesis, the phrase *ideal partitioning strategy* is used as it applies to the specific problem of parallel discrete event simulation (PDES) for large VHDL circuits using VSIM. It is reasonable to expect that such a partitioning strategy will be equally applicable to other parallel problems that have similar static task dependency characteristics. The desirable properties of an ideal parallel VHDL partitioning strategy include the following:

- *Computational Efficiency* - The partitioning algorithm should be computationally efficient, requiring only polynomial time to converge to a *good* solution. Finding the *optimal* solution[4] to the general mapping problem has been shown to be NP-complete, thus rendering it computationally infeasible to seek such a solution for large and complex problems (17:63, 22:142). Parallel algorithms are one potential means of achieving the necessary computational efficiency, although numerous sequential algorithms have been proposed as well.

---

[3] Throughout this thesis, the terms *behavior*, *process*, and *task* are used interchangeably to represent the vertices of the problem-graph.

[4] In the context of this thesis, the *optimal* solution is defined as the mapping that results in the fastest simulation for a given number of processors. A *good* solution is defined as any solution that results in "near-optimal" simulation run times.

- *Balanced Workload* - The partitioning algorithm should result in a balanced computation load among all available processors (5:294). This requires that the percentage of time spent performing useful computations be approximately equal for each processor.
- *Exploitation of Inherent Parallelism* - The partitioning algorithm should produce solutions which take advantage of the parallelism inherent in the simulation application.
- *Minimized Inter-Processor Communications* - The partitioning algorithm should produce solutions with minimal communications between processors (5:294). The two primary factors to consider here are the number of communication links between tasks on different processors, and the relative frequency with which messages are sent over those links.
- *Scalability* - The partitioning algorithm should be easily scalable, both in terms of the number of tasks in the problem graph, and the number of processors (5).
- *Deterministic Solutions* - The partitioning algorithm should produce deterministic solutions which are based upon the known static inter-dependency structure of the problem graph.
- *Input Problem Variations* - The partitioning algorithm should be applicable to a wide variety of VHDL circuits, including those with feedback loops.
- *Accounts for PDES Synchronization Protocol* - Ideally, the partitioning algorithm should be equally applicable regardless of the particular parallel discrete event simulation protocol used. However, this is not feasible since the simulation protocols play a major role in defining the amount of inter-processor communications. Instead, given a specific simulation synchronization protocol,

the partitioning algorithm should account for its overhead requirements when making decisions regarding task partitioning.

2.4 *General Approaches to the Mapping Problem*

Numerous approaches to the static task mapping problem have been pursued including algorithms based on graph theory, mathematical programming, queueing theory, and various heuristic approaches such as simulated annealing (21:1409). Two specific graph-based models which have been proposed for modeling the static task allocation problem involve the use of a *task precedence graph* (TPG) and a *task interaction graph* (TIG). The task precedence graph model consists of a directed graph in which the vertices represent tasks and the edges represent inter-task execution dependencies. Computational and communication costs are represented by adding weights to the vertices and edges of the graph. A task interaction graph has the same basic structure. However, a task precedence graph models execution precedence dependencies whereas a task interaction graph models the need for inter-task communication without explicitly representing such temporal dependencies. All tasks are considered independently and concurrently executable. In both models, the goal is to map the tasks to processors so as to minimize the total program execution time (21:1409).

The class of parallel problems that can be modeled by a task interaction graph (TIG) consists primarily of iterative algorithms in which all tasks can execute independently during each iteration, and exchange data values only in between iterations (21:1409). Many algorithms for matrix manipulations fit into this category.

On the other hand, problems which exhibit temporal dependencies (e.g., task B cannot execute until task A has executed) between tasks can be modeled with the task precedence graph (TPG). The temporal dependencies modeled by the directed arcs of the

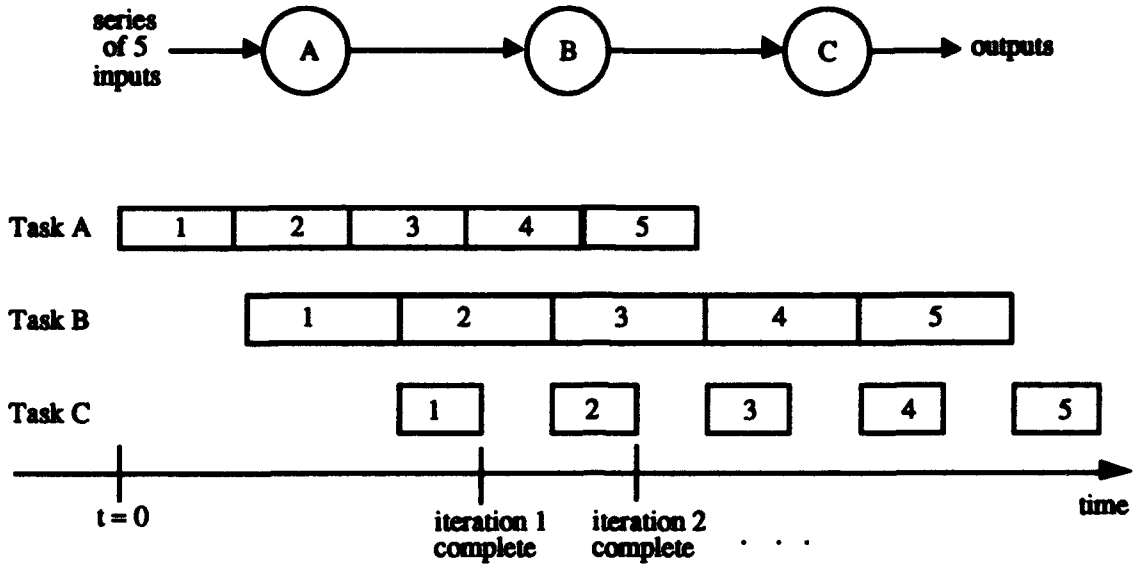


Figure 2. Iterative PDES Algorithm

TPG define a series of tasks that must be executed sequentially, making these problems inherently difficult to parallelize.

The set of iterative problems encompassing parallel discrete event simulation (PDES) can be represented by the TPG model by considering the temporal dependencies in terms of a single iteration, and overlapping the iterations in a pipeline fashion. Figure 2 shows an example for three simple tasks over five iterations, with the assumption that each task is on a separate processor. Task A produces a series of five data values, each of which is acted on separately by task B. In turn, task B produces a series of five data values, each of which is acted on separately by task C. Because task A has no dependencies, it can run independently to completion. However, task B cannot begin its operation on the first iteration until the first input has been received from task A. A similar relationship holds between tasks B and C. For illustrative purposes, each task requires a different amount of time to perform its computation on the data values flowing through the system as indicated in the figure. The numbers in the rectangles represent the iteration that each task is on at a given point in time, with time not covered by a rectangle representing idle time for that task. Note that because this is an event-driven simulation, the three tasks are not

necessarily executing in lockstep. For example, task A begins its third iteration before task B completes its first iteration.

The TPG model is used throughout this thesis to model the VHDL mapping problem[5] by modeling the individual VHDL behaviors as graph vertices and the inter-behavior dependencies as directed edges. Further discussion of this model can be found in the requirements section of chapter 3. The remainder of this chapter examines numerous aspects of various partitioning schemes using graph-theoretic techniques that have been proposed for a variety of parallel problems.

2.4.1 Random Partitioning. *Random Partitioning* involves the random distribution of the tasks into the desired number of LPs, and was the partitioning scheme used for prior AFIT research on circuits with more than 100 behaviors (4). It is one of the simplest partitioning algorithms, but potentially the most ineffective. Under this approach, only the load balancing among the LPs is considered. The behavior dependencies, and associated communications costs, are ignored. Breeden shows that in some limited circumstances, speedup can be obtained with this partitioning scheme (4:70). However, because the behavior dependencies are not considered, the resulting partition has a large number of inter-behavior dependencies that cross the partition boundaries, and often has artificial feedback imposed upon it. The situation worsens as the number of LPs grows. As a result, this partitioning strategy is not likely to scale very well or provide very good performance. This conclusion is supported by the limited data available from previous research (4:70).

2.4.2 Simple Data Partitioning. A slightly better algorithm, but just as simple as the random partitioning, is referred to as *Simple Data Partitioning* (SDP) (9:78). Under

[5] The *VHDL mapping problem* in this thesis is considered relative to the VSIM/SPECTRUM simulation environment.

the SDP approach, each vertex in the graph has a weight associated with it, with the weight calculated as the degree of the vertex (number of arcs to and/or from the vertex). Each partition is filled one at a time, with vertices selected for inclusion by decreasing order of their weight until the combined weight of the current partition is approximately equal to the calculated average weight (9:78). The result is that vertices with a high in/out degree will tend to be grouped together in partitions with fewer vertices. The method of selecting which of several vertices with the same weight is not specified, and is assumed to be arbitrary. As a result, it is reasonable to expect that for graphs in which a large portion of the vertices have equal weights, results similar to those for random partitioning would be achieved.

Since each arc in the graph represents a potential for inter-task communications, a vertex with a high in/out degree is likely to have more inter-task communications than a vertex with a small in/out degree. Thus, on the surface it seems as though grouping vertices with a potential for large amounts of communications in the same partition would tend to minimize inter-partition communications. The fallacy of this approach is that a group of vertices with high in/out degree may in fact result in a large amount of communications, but not necessarily with other vertices in the same partition.

2.4.3 General Graph Contraction & Layout Algorithm. An alternative approach to the mapping problem attempts to address both cardinality variation and topological variation at the same time by modeling a given parallel algorithm with a *family* of graphs $\{G_n\}$. Each graph G_n represents the static dependency graph of a parallel algorithm for a problem of size n. A similar graph G_p models the physical processors and interconnection structure of the target architecture for P processors (2:307, 3:441). Given a problem of size n represented by graph G_n, the proposed approach involves *contracting* the graph G_n into a smaller graph G_k of size k from the same graph family. The contraction process is

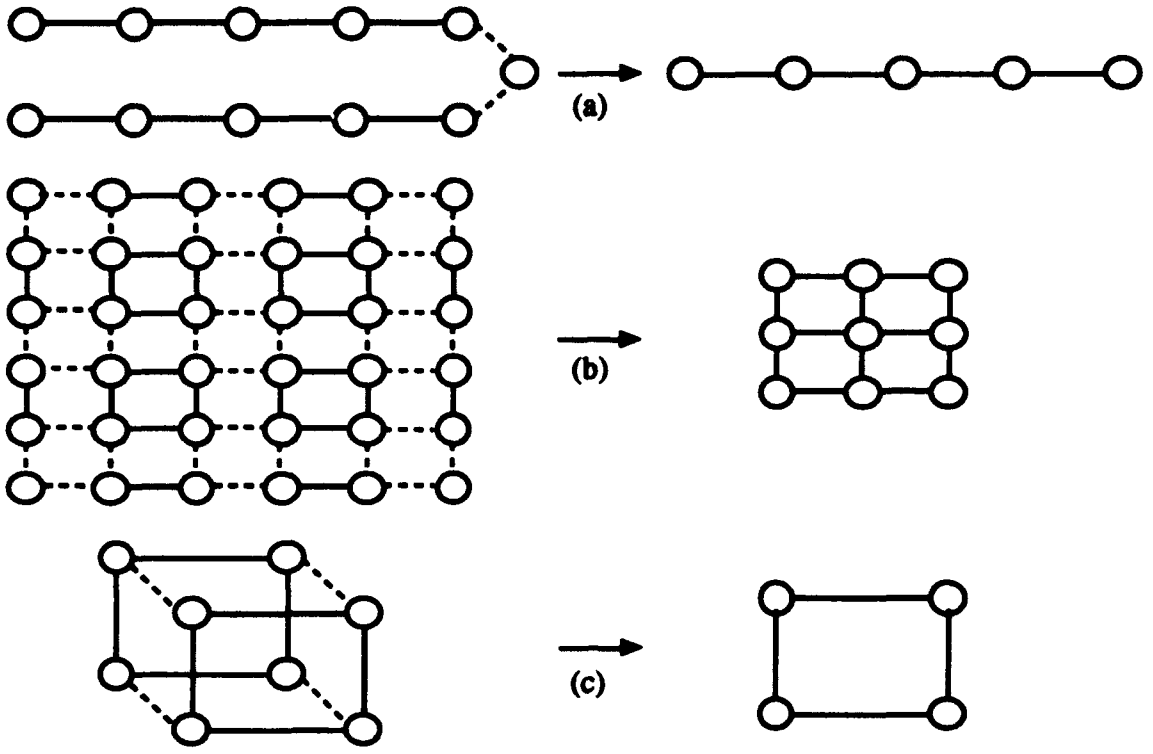


Figure 3. Contraction within a graph family. Vertices incident to dashed edges are grouped into a single vertex (3:449)

continued until k = P, thereby eliminating the cardinality variation. The next step is to lay out the contracted graph G_k onto the physical interconnection graph G_p, thereby eliminating the topological variation. The final step in the algorithm uses multiplexing to implement the problem-graph G_n on the image of G_k (2:307, 3:441). Three examples of contracting an algorithm represented by G_n into a graph from the same family G_k are shown in Figure 3.

In the specific case of the VSIM/SPECTRUM simulation environment, this approach could be improved by encapsulating each vertex on the contracted graph G_k inside of a single logical process (LP), thereby avoiding the communications and context switching overheads associated with multiplexing multiple tasks on a single processor. Even then, this approach makes two implicit assumptions about the problem domain that severely limits its feasibility as a solution to the mapping problem for parallel VHDL simulation.

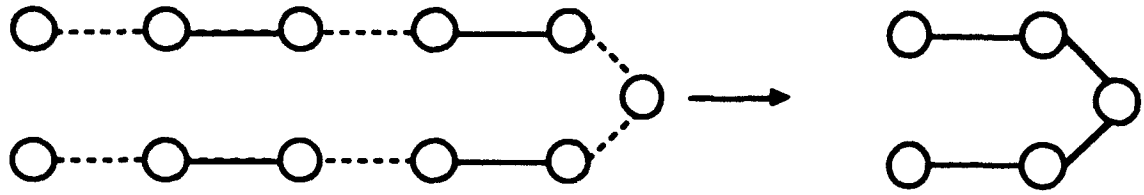


Figure 4. Alternative Contraction for Graph of Figure 3.a

First, it assumes that the dependency graph of the parallel algorithm (in this case the simulation of a structural VHDL circuit), will exhibit a pattern that remains consistent as the problem size grows larger. Second, it assumes that there exists a graph $G_k \in \{G_n\}$ whose cardinality and topological layout are the same as that of the physical interconnection graph G_p.

As a final observation concerning this approach, it should be noted that even if both of these assumptions hold and this methodology can be used to map the problem-graph onto the processor-graph, it may or may not result in the most efficient inter-processor communications structure. If tasks on the same processor are encapsulated within a single LP, external messages will not be required for two such tasks to communicate. Therefore, each dashed edge in Figure 3 represents a potential inter-processor communications link that has been eliminated in the contracted graph. Considering the 8-node graph of Figure 3.c which is contracted into 4 LPs, there is clearly no possible way that this contraction could be done such that more than four edges are eliminated without sacrificing load balancing. However, consider the 11-node graph of Figure 3.a which is contracted into 5 LPs with the elimination of only two edges. Figure 4 shows an alternative contraction that would eliminate six edges, thus reducing the potential inter-processor communications.

2.4.4 Strip Assignment Algorithm. Two other approaches, *Strip Assignment* and *Two-Dimensional Mapping*, have been proposed for the specific problem of mapping metalforming applications using finite element methods onto a hypercube architecture

(22, 21). Both of these heuristic approaches attempt to address the load-balancing and inter-processor communications aspects of the mapping problem separately. Load-balancing is addressed by attempting to allocate an equal number of tasks to each processor. This is analogous to allocating an equal number of VHDL behaviors to each Logical Process (LP), with one LP per processor. The inter-processor communications overhead is addressed in two ways. First, the algorithms attempt to group tasks together on the same processor such that the amount of inter-processor communications required is minimal. Second, each algorithm attempts to allocate the task groups among the processors such that only nearest-neighbor communications are ever required (22, 21).

The strip assignment method evenly distributes the tasks among the available processors in such a way that each processor will only need to communicate with no more than two immediately adjacent neighbor processors. Letting N be the total number of tasks in the problem-graph and P be the total number of processors, the number of tasks per processor to achieve a balanced load is $\lceil N/P \rceil$ for some processors, and $\lfloor N/P \rfloor$ for the remainder (22:144, 21:1414). An example for N = 48, P = 6 is shown in Figure 5. Letting N_C be the maximum number of tasks in any column of the problem-mesh, and N_R be the maximum number of tasks in any row, the order in which tasks are added to a partition depends on the relative magnitudes of N_C and N_R. Beginning at any corner of the problem mesh, tasks are added to the current partition along the columns if $N_C \leq N_R$, or along the rows if $N_R \leq N_C$. Subsequent partitions begin where the previous one left off (22:145).

The strip assignment method assumes that the graphical representation of the problem can be represented by a two-dimensional mesh layout in which each task has at most four communication paths to its nearest neighbor tasks. In addition, in order for the strip method to guarantee that nearest-neighbor communications are maintained among the

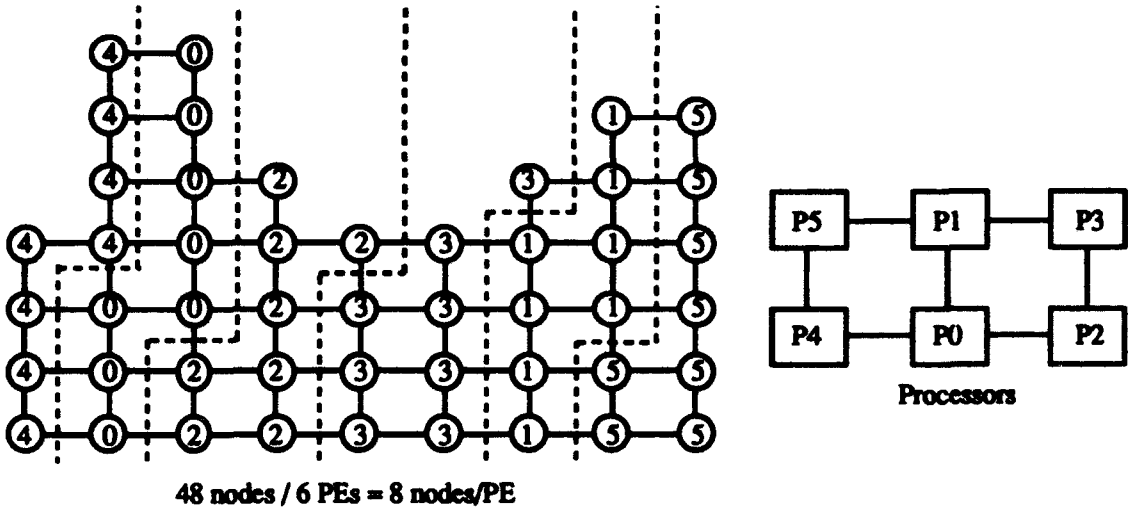


Figure 5. Example of the Strip Assignment Method for a Problem-Mesh (22:143)

physical processors, a certain constraint on the problem-mesh must be met (22:144, 21:1414). Specifically, the strip method requires that:

$$\lfloor N/P \rfloor > \min (N_C , N_R)$$

It is reasonable to expect that very few, if any, VHDL circuit dependency graphs will ever meet the two-dimensional mesh layout requirement. Thus, this algorithm does not seem suitable for the partitioning of structural VHDL circuit simulations.

2.4.5 Two-Dimensional Mapping Algorithm. The other method that has been applied to metalforming applications using finite element methods is *Two-Dimensional Mapping* (22:145). This approach differs from the *Strip-Assignment* method in that an attempt is made to reduce the number of nearest-neighbor communication links between processors. As can be seen from Figure 5, the *Strip-Assignment* method results in partitions that span the entire problem-mesh (either column-wise or row-wise). As a result, a large number of communications links in the problem-mesh cross the partition boundaries. The *Two-Dimensional Mapping* approach capitalizes on the fact that square partitions will have a smaller perimeter-to-area ratio than the "rectangular" partitions of

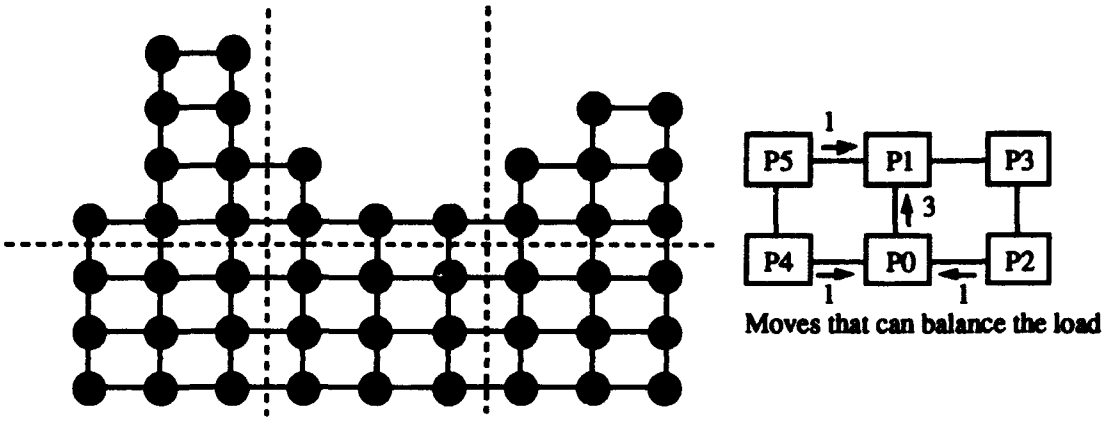


Figure 6. Initial Partition of a Two-Dimensional Mapping for a Problem-Mesh

the *Strip-Assignment* method, and thus, should result in fewer communications links crossing the partition boundaries. Schwan et al. suggest a three-step approach (22:145):

- Divide the problem-mesh into square partitions as if the problem-mesh were perfectly regular. Depending on the dimensions of the problem mesh, some partitions may only approximate squares.
- Use a border-refinement algorithm to account for an irregular problem-mesh and achieve load-balancing.
- Use a secondary refinement algorithm to attempt further minimization of inter-processor communications while maintaining a balanced load. This step is optional.

Figures 6 and 7 demonstrate a two-dimensional mapping for the same problem-mesh as in Figure 5. First, Figure 6 shows the initial two-dimensional partition consisting of simple horizontal and vertical lines through the problem-mesh. Visualizing the problem-mesh as being perfectly regular (all columns have N_C processes and all rows have N_R processes), the lines are placed so that the resulting partitions are as close to being identically sized squares as possible. However, because the problem-mesh is not perfectly regular, the resulting partitions are not evenly balanced. The second step in the algorithm requires moving processes from one partition to a neighboring partition until all partitions

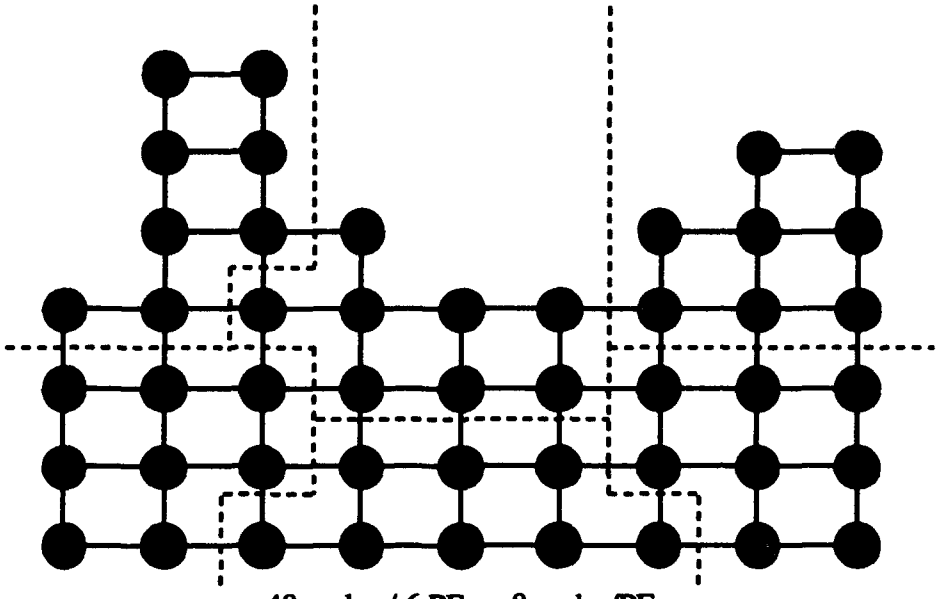


Figure 7. Two-Dimensional Mapping Partitions after Border-Refinement

are balanced. The resulting partitions are shown in Figure 7. Note that in the strip-assignment partitions of Figure 5, there were 32 inter-partition communication links. In the two-dimensional partition, the number of inter-partition communication links has been reduced to 21 while maintaining a balanced load.

An additional benefit of two-dimensional mapping over strip-assignment is that the former does not impose any constraints on the number of processors P as it relates to the dimensions of the problem-mesh N_R and N_C .Thus, it can be applied to problem-meshes where strip-assignment is not applicable. Like strip-assignment, however, this approach is only applicable to problems that can be represented as a two-dimensional mesh. Thus, as it is presented here, two-dimensional mapping does not seem suitable for the partitioning of structural VHDL circuit simulations. However, the idea of performing load balancing and reducing the inter-partition communication costs by refining the partition boundaries can be extended to algorithms that *can* be applied to other forms of problem graphs.

2.4.6 Algorithm M, An Optimal Approach. Another approach to the mapping problem, *Algorithm M* , has been proposed for static assignment of tasks in a distributed system in which the processors communicate over an ethernet-based medium (16). In such a system, all processors communicate over a shared pathway for which all processors must compete. As a result, the processor interconnection graph G_p is fully connected, and no special steps are required to layout the partitioned problem-graph G_k onto the processor graph G_p (16:240).

Lo argues that the goals of a static partitioning algorithm for a distributed system are different from those of a static partitioning algorithm for a parallel system given an identical problem domain (16:240). However, in both systems, the inter-processor communications (IPC) should be minimized while meeting some load balancing constraint (e.g. equal number of processes per processor). The two systems differ primarily in the relative cost of inter-processor communications in relationship to the cost of some load imbalance. However, these relative cost differences also exist between different classes of parallel systems, and even between different applications on the same parallel system. Thus, the goals of partitioning problem-graphs for distributed systems and partitioning them for parallel systems are actually the same. It is only the mapping of the partitions onto the physical processors that differs, and then only if one is concerned about maintaining nearest-neighbor communications in parallel systems. As a result, an effective partition for one type of system is likely to be just as effective on the other type of system. Given this fact, Algorithm M can be studied as it might apply to partitioning problem-graphs for parallel systems.

Algorithm M has been shown to provide an optimal partitioning of a problem-graph G_n onto P identical processors in polynomial time providing that the number of tasks n in G_n is no more than twice the number of processors ($n \leq 2P$), and providing that no more than two tasks can be assigned to any single processor (16:241). Although this restriction

makes this approach unsuitable for parallel VHDL simulations which may have hundreds to tens of thousands of tasks per processor, it is discussed here because it leads into the sub-optimal heuristic approximation discussed in the next section which removes the restriction on n.

Algorithm M begins by finding a *maximum weight matching* of the problem graph G_n. A *matching* is defined as a set of edges from a graph such that no two edges in the set share a common vertex. The sum of all the weights of the edges in the matching forms the *weight of the matching*. The *maximal weight matching* is the matching of the graph with the largest weight. Algorithms exist to find the maximal weight matching of a graph in polynomial time (16:241).

After the maximal weight matching has been found, the next step is assigning the two tasks corresponding to each edge in the matching to a processor with no other tasks assigned to it (16:241). It should be noted that a maximal weight matching may not necessarily contain all of the tasks in the problem-graph. Tasks that are not connected by an edge in the matching are arbitrarily paired and assigned to a processor with no other tasks assigned to it. Finally, if there remains a single unpaired task, it is assigned by itself to any remaining processor that has no other tasks assigned to it. Lo states that this algorithm will provide an optimal partition for a given input problem-graph that meets the constraint $n \leq 2P$ (16:241). If the problem is such that communications costs between some pairs of tasks will be greater than between other pairs of tasks (due to frequency of messages, size of messages, etc.), this algorithm has the interesting property that those pairs of tasks with the highest communications costs will be assigned to the same processor where communications costs are negligible.

As mentioned previously, the restriction $n \leq 2P$ limits the class of problem-graphs that may be partitioned using this algorithm. An additional shortfall of the algorithm is the implicit assumption that each task will have an equal weight (in terms of computational

cost). Depending on the extent to which this may not be true, this further limits the class of problems for which this algorithm will result in a partition that is truly optimal in terms of both inter-processor communications *and* load balancing.

2.4.7 Algorithm H, A Heuristic Approximation of Algorithm M. Algorithm H represents a sub-optimal approximation of Algorithm M for the polynomial-time partitioning of an arbitrary number of tasks n among P processors with a bound B on the maximum number of tasks per processor where $\lceil n/P \rceil \leq B \leq n$ (16:242). This is accomplished by first reducing the original problem graph G_n with n tasks to a smaller graph G_k with k nodes using a "Sort Greedy Algorithm" so that $k \leq 2P$ and with G_k having no more than $\lceil B/2 \rceil$ tasks per node (16:243). An optimal partitioning for the reduced graph G_k can then be obtained using Algorithm M. However, this partition may not necessarily be optimal for the original problem-graph G_n (16:242).

Lo's simulation results have shown that Algorithm H performs relatively well, finding an optimal partition in over 80% of the test cases run (16:243). However, this data was collected on only a small set of problem-graphs with no more than 35 tasks and 5 processors. In addition, because a greedy-type algorithm is used for the initial graph reduction from G_k to G_n, poor assignments may result when the problem-graph contains relatively uniform communication costs (16:243). This is likely to be the case in large structural VHDL circuit simulations. Nevertheless, the concept of a phased approach to partitioning a problem-graph presented by this algorithm holds potential for a polynomial time general-purpose partitioning algorithm that will provide near-optimal solutions.

2.4.8 Depth-First Breadth-Next Algorithm. An algorithm called *Depth-First Breadth-Next* (DFBN) has been proposed to partition problem dependency graphs (17). The goals of this algorithm are to assign dependent tasks to the same processor, and

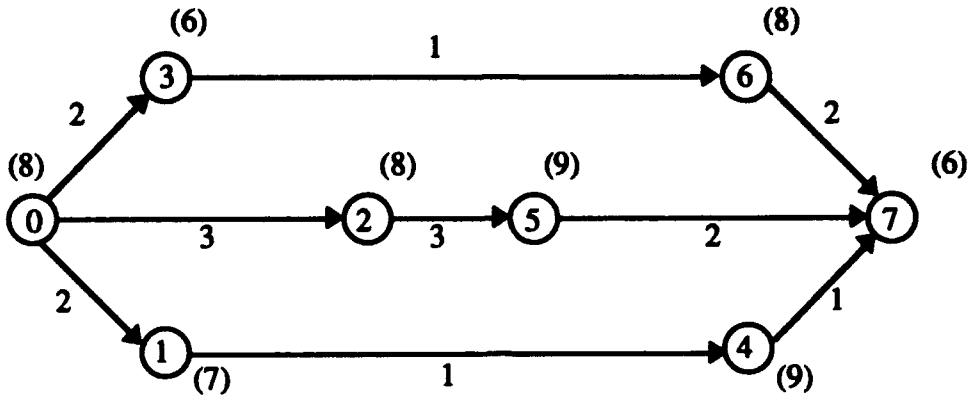


Figure 8. Example Problem Graph for DFBN Partitioning (17:64)

independent tasks to different processors (17:63). The name is derived from the manner in which the problem graph is traversed when partitioning dependent tasks.

Two assumptions concerning the set of applicable problem-graphs are made for the DFBN algorithm. First, it is assumed that the graphs are acyclic. Second, it is assumed that the tasks execute non-preemptively, and must run to completion once they begin execution (17:65). In the example problem graph of Figure 8, each edge is labeled with a total communications time, and each task is labeled with a total execution time. These values correspond the weights of the edges and tasks respectively, and are used in prioritizing non-critical tasks on the same processor for scheduling purposes (17:69). A critical task is one that must be executed before any others in order for any progress to be made (e.g., task 0 is critical).

The actual DFBN partition ignores the task and edge weights, and considers only the interdependency relationships between the tasks. Since the edges in the graph represent temporal dependencies and the assumption has been made that the tasks run non-preemptively to completion, simultaneous execution of tasks located on the same path is not allowed. Thus, each path in the graph represents a set of dependent tasks, and tasks that are not on the same path are independent (17:68). The goal of assigning dependent tasks to the same processor may not be completely achievable since two independent

tasks may have common dependencies (17:67). For example, in the graph of Figure 8, tasks 1, 2, and 3 are all independent, but all share the common dependency on task 0. Nevertheless, it may be possible to achieve a good approximation of the stated goal.

The partitioning algorithm simply performs a depth-first search of the problem-graph, beginning at the source nodes. When a task has been discovered by the search, it is marked so that it will not be included as part of more than one path. According to the DFBN algorithm, each distinct path in the problem-graph resulting from the DFBN search is assigned to a different processor (17:68). However, this assumes that the number of distinct paths will be less than or equal to the number of available processors, which may not be true. Furthermore, the DFBN algorithm makes no effort to balance the computation load among the processors. Some progress is made towards limiting inter-processor communications since communications between dependent tasks of the same path are on the same processor. However, this minimization is likely to be unevenly applied and far from optimal.

2.4.9 Kernighan-Lin Algorithm. A graph-partitioning procedure proposed in 1970, known as the *Kernighan-Lin* algorithm after its authors (14), is often a standard used by others for comparison with their own partitioning algorithms (9:78). The *Kernighan-Lin* algorithm divides a weighted problem-graph into partitions of equal cost and with a minimum cutset. This is accomplished by first making an arbitrary, but even, partition of the problem-graph. Tasks are then swapped between partitions until a minimum cutset is found, when the partition is locally minimum (14:295). Depending on the nature and size of the problem-graph and the initial partition, the resulting cutset may also be globally minimum, resulting in an "optimal" partition (14:295, 9:78).

2.4.10 Simulated Annealing. *Simulated Annealing* (SA) refers to an algorithm that is used to model groups of atoms being cooled to a ground state, using the concept of a decreasing temperature to aid in the convergence of a solution (9:79). When applied to the mapping problem, *Simulated Annealing* begins with a random initial partition of the problem-graph, and involves moving tasks to other partitions if the total cost of the partition is lowered. The partition cost equation has factors for the inter-partition communication costs and the relative cost of the resulting load-imbalance among the partitions. A random number generator is used to select both the process to move and the desti: .tion partition. If the move does not result in a lower total cost for the partition, the move may still be made based on a decreasing random probability function (9:78).

The results of Conrad and Agrawal have shown that *Simulated Annealing* solutions to the mapping problem approach optimal solutions more often than the *Kernighan-Lin* algorithm, but require approximately two orders-of-magnitude more time to reach a solution (9:78). Another shortfall of this approach is its non-determinism which is due to the use of a random-number generator to select the proposed moves and to control the probability function, as well as the randomness of the initial partition. Depending on the initial partition and the sequence of proposed moves selected by the random number generator, it is possible to get different results for the same input.

2.4.11 Mean Field Annealing. A related algorithm to *Simulated Annealing*, called *Mean Field Annealing* (MFA), also refers to an algorithm that models groups of atoms being cooled to a ground state (9:79). As in *Simulated Annealing*, MFA begins with a random initial partition and uses a random number generator at each iteration of the algorithm to select a task for consideration. The state of the selected task is updated to determine the proper partition assignment. To control the moves and force a convergence to a solution, the algorithm uses a probability function that takes into account the total

cost of the partition and decreases according to some specified schedule (5). The main advantage of this algorithm is that it converges to a solution much faster than *Simulated Annealing*, requiring an execution time on the same order as the *Kernighan-Lin* algorithm (9, 5).

Derivation of the MFA algorithm is accomplished by analogy to the Ising spin model which estimates the state of a system of particles, or *spins*, in a state of thermal equilibrium by computing an energy function (5:295). When applied to the mapping problem, the energy function is interpreted as the cost of a given partition. This cost function can be computed by an objective function H containing a factor for the inter-partition communications H_c, and a factor for the partition load imbalance H_b (9:78):

$$H = H_c + \alpha H_b$$

where the parameter α is a scaling factor used to maintain a balance between the objectives of minimizing communication costs and maintaining a balanced computation load (5:297, 9:78). The factor α can be taken as an input parameter to control the level of load imbalance that is acceptable in a partition.

A function `spin(i,p)` is defined, with output s_{ip}, which returns the probability of mapping task i to processor p. By definition, s_{ip} is a continuous variable in the range $0 \leq s_{ip} \leq 1$. However, as the algorithm reaches a solution, the spin values s_{ip} converge to either a 0 or a 1, with a 1 indicating that task i is mapped to processor p (5:296). Thus, the solution can be represented by an N x P spin matrix with each row containing P-1 zero entries. An example for N = 8 and P = 4 is shown in Figure 9.

The cost function H is defined by (5:296):

$$\begin{aligned} H &= H_c + \alpha H_b \\ &= \frac{1}{2}\sum_{i=1}^{N}\sum_{j\neq i}\sum_{p=1}^{P}\sum_{q\neq p} e_{ij}\, s_{ip}\, s_{jq}\, d_{pq} + \frac{\alpha}{2}\sum_{i=1}^{N}\sum_{j\neq i}\sum_{p=1}^{P} s_{ip}\, s_{jp}\, w_i\, w_j \end{aligned}$$

where e_{ij} represents the communications cost between tasks i and j, w_i represents the computation cost of task i, and d_{pq} represents the relative communications cost per

		P processors			
		1	2	3	4
	1	0	0	0	1
	2	0	0	0	1
N	3	0	1	0	0
processes	4	1	0	0	0
	5	0	0	1	0
	6	0	1	0	0
	7	1	0	0	0
	8	0	0	1	0

Figure 9. Example Spin Matrix for N = 8 and P = 4 (5:296)

message between processors p and q. In order to calculate the function `spin(i,p)`, the mean field function ϕ_{ip} is defined as (5:296):

$$\phi_{ip} = -\sum_{j \neq i}^{N} \sum_{q \neq p}^{P} e_{ij} s_{jq} d_{pq} - \alpha \sum_{j \neq i}^{N} s_{jp} w_i w_j$$

Each individual spin average s_{ip} is proportional to $e^{\phi/T}$ where T is the *temperature* of the system, and s_{ip} is normalized as (5:296):

$$s_{ip} = \frac{e^{\phi_{ip}/T}}{\sum_{q=1}^{P} e^{\phi_{iq}/T}}$$

This normalization forces each row of the spin matrix to sum to 1, and ensures that each task i is mapped to only one processor when the system stabilizes for a given temperature T (5:296). During each iteration of the MFA algorithm, a randomly selected row i in the spin matrix is recalculated using the equations for ϕ_{ip} and s_{ip}. After each iteration, the cost function H is recalculated in order to detect a convergence to an equilibrium state for a given temperature T. If the cost function H does not decrease after a specified number of iterations, the system is said to be stabilized for the current temperature. The temperature T is then decreased according to a specified schedule (5:297). As T is decreased further, the probability that the randomly selected task i will be moved from its currently assigned processor decreases, and a final solution is eventually reached.

The results of Bultan and Aykanat using the MFA algorithm to solve the mapping problem have shown that its solutions approach the quality of those from the *Simulated Annealing* algorithm in 1/20th the time (5:301). However, like *Simulated Annealing*, the MFA algorithm has the undesirable property that it is non-deterministic due to the random initial partition, and the random selection of processes to update from the spin matrix. Nevertheless, it appears to be a viable alternative to solving the mapping problem for general parallel problems.

2.5 Summary

The diversity of methods for solving the general mapping problem outlined in this chapter represent only a small cross-section of those possible. None of the methods presented claim to provide an optimal solution for all problems. However, most of the methods provide *good* solutions for a specific subset of problems.

The next chapter presents a partitioning algorithm that draws upon properties from several of the algorithms presented in this chapter in an attempt to find a scheme that provides consistently *good* partitions for a wide variety of structural VHDL circuit simulations.

III. Problem Analysis

3.1 Overview

This chapter presents a discussion of the graph-partitioning algorithm proposed in this thesis for mapping VHDL circuit simulations to multiple logical processes for execution in parallel. In order to understand many of the decisions made in formulating the partitioning algorithm, it is necessary to first understand the target application. Therefore, the chapter begins with a review of the implementation of AFIT's parallel VHDL simulator, VSIM, as implemented by Comeau and Breeden.

3.2 Implementation of VSIM

Previous AFIT research has resulted in the successful implementation of a parallel simulator for simulating structural VHDL circuits on an Intel hypercube (8, 4). This simulator, known as VSIM, uses the intermediate C source code created by the sequential Intermetrics commercial simulator as shown in Figure 10 (4:21). To simulate a structural VHDL circuit using VSIM, the Intermetrics sequential simulator compiles the VHDL source code and creates an IVAN (Intermediate VHDL Attributed Notation) file containing intermediate-form code descriptions of the circuit components. Next, during the model generate phase, the Intermetrics simulator transforms the IVAN file into C source code files and creates the corresponding object files (8). Using a tool called `pbuild`, these C source code files (and their associated header files) are intercepted and transformed into VSIM compatible files that can be executed in parallel (4:20).

VSIM is implemented to run over the SPECTRUM parallel simulation testbed which manages the inter-process synchronization (4, 20). VSIM itself is independent of the parallel discrete event simulation (PDES) synchronization protocol being used. To

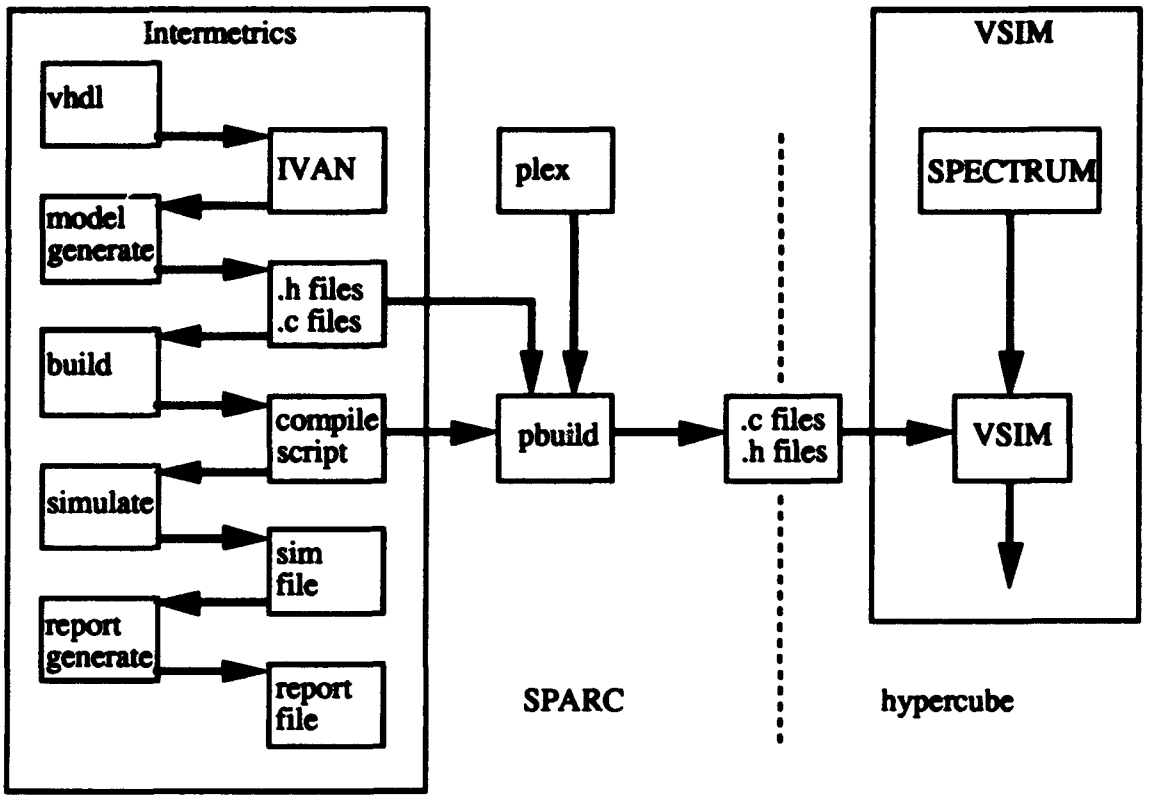


Figure 10. VSIM Parallel Simulation Session (4:21)

maintain continuity with prior AFIT research, this research uses the conservative Chandy-Misra synchronization algorithm with null messages[6] to provide deadlock avoidance (7).

The structural descriptions and simple processes representing the VHDL subset implemented by VSIM form "behavioral instances" which are partitioned into groups to form Logical Processes (LPs) (4:21). The LPs are in turn partitioned among the available processor nodes for parallel execution. Each LP contains a copy of the necessary application code, the complete SPECTRUM code, and the machine dependent operating system interface code (12). As shown in Figure 11, the SPECTRUM testbed provides an interface between the application code and the machine dependent interface to the operating system. As such, the SPECTRUM code provides the message sending and receiving functionality by which two LPs communicate with each other. Variants of the

[6] Null messages contain no signal change information. They exist for the sole purpose of advancing the simulation clock, and do not correspond to actual events in the physical system (7).

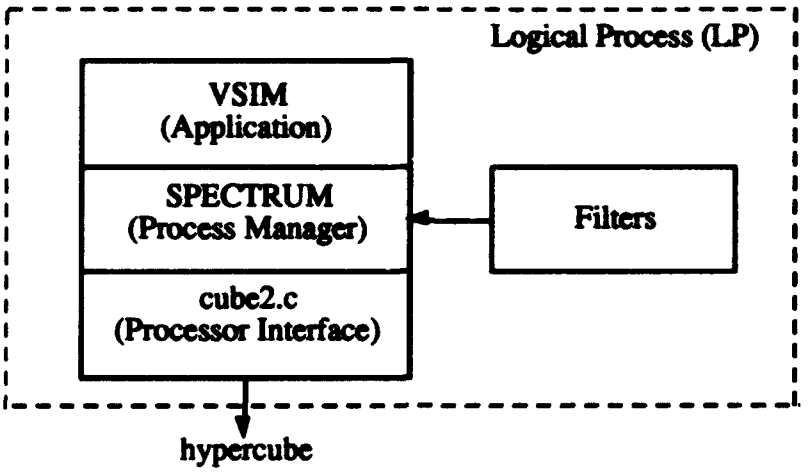


Figure 11. SPECTRUM Interface for a Single LP (4, 12)

simulation protocol used to maintain synchronization are implemented via *filters* as shown in Figure 11 (4, 12). In the case of VSIM, the Chandy-Misra null-message protocol is implemented via a filter called `vhdlclocks`.

In addition to running in parallel on an Intel hypercube, VSIM also has a sequential mode which can be executed on a Sun Sparcstation. In fact, VSIM's parallel simulation cycle is a direct extension of its sequential simulation cycle. Furthermore, prior to executing VSIM on the hypercube, the circuit must be executed in the sequential mode for at least one entire simulation cycle in order to extract the behavior numbers and inter-dependency relationships. Therefore, this section begins with a discussion of the sequential mode of VSIM. This is followed by an introduction to the SPECTRUM parallel simulation testbed, which leads into a discussion of how the sequential simulation cycle is extended in VSIM's parallel mode.

3.2.1 Sequential Simulation.

3.2.1.1 Data Structures. The four fundamental data structures used in VSIM's sequential mode are directly derived from the commercial Intermetrics simulator and are as follows (4:21-23):

- **Behavioral Instances** - Used to describe the input/output behavior of each executable VHDL process. A separate behavioral instance exists for each VHDL behavior in the simulation, although a single execution routine may be shared by several instances (4:21).
- **Signal Record Table** - Used to maintain the current state of each signal in the simulation including pointers to behavioral instances in order to identify the dependency connections for each signal (4:21-22).
- **Behavior List** - Used to maintain a list of all behavioral instances scheduled for execution during the current simulation time. All behaviors are scheduled for execution (i.e., placed in the behavior list) at simulation startup (t = 0) in order to initialize the values of all signals. When a behavior is executed, it is removed from the behavior list. A change in a signal value will cause all dependent behaviors to be re-scheduled for execution (4:22).
- **Active Record List** - Used as a next-event list for the simulator. An "event" is defined as the output of a behavioral instance execution routine that *may* result in a signal change. Entries in the list are maintained in increasing order of the simulation time associated with each scheduled event (4:23).

Figure 12 shows an example of the interrelationship of these fundamental data structures. In this example, signal id 2 has an entry on the active record list because it is changing values from a '0' to a '1' at simulation time 50. The active record list entry contains the new value and a pointer to the specific signal record. The signal record contains pointers to the signal's current value in main memory and the affected behavior instances - behaviors 2 (AND gate) and 3 (XOR gate) in this example. These affected behaviors are in turn added to the behavior list for execution at time 50 (4:23, 8:3-12). Note that the entries in the behavior list contain pointers to the associated behavioral instances which in turn contain pointers to the appropriate behavioral execution routine.

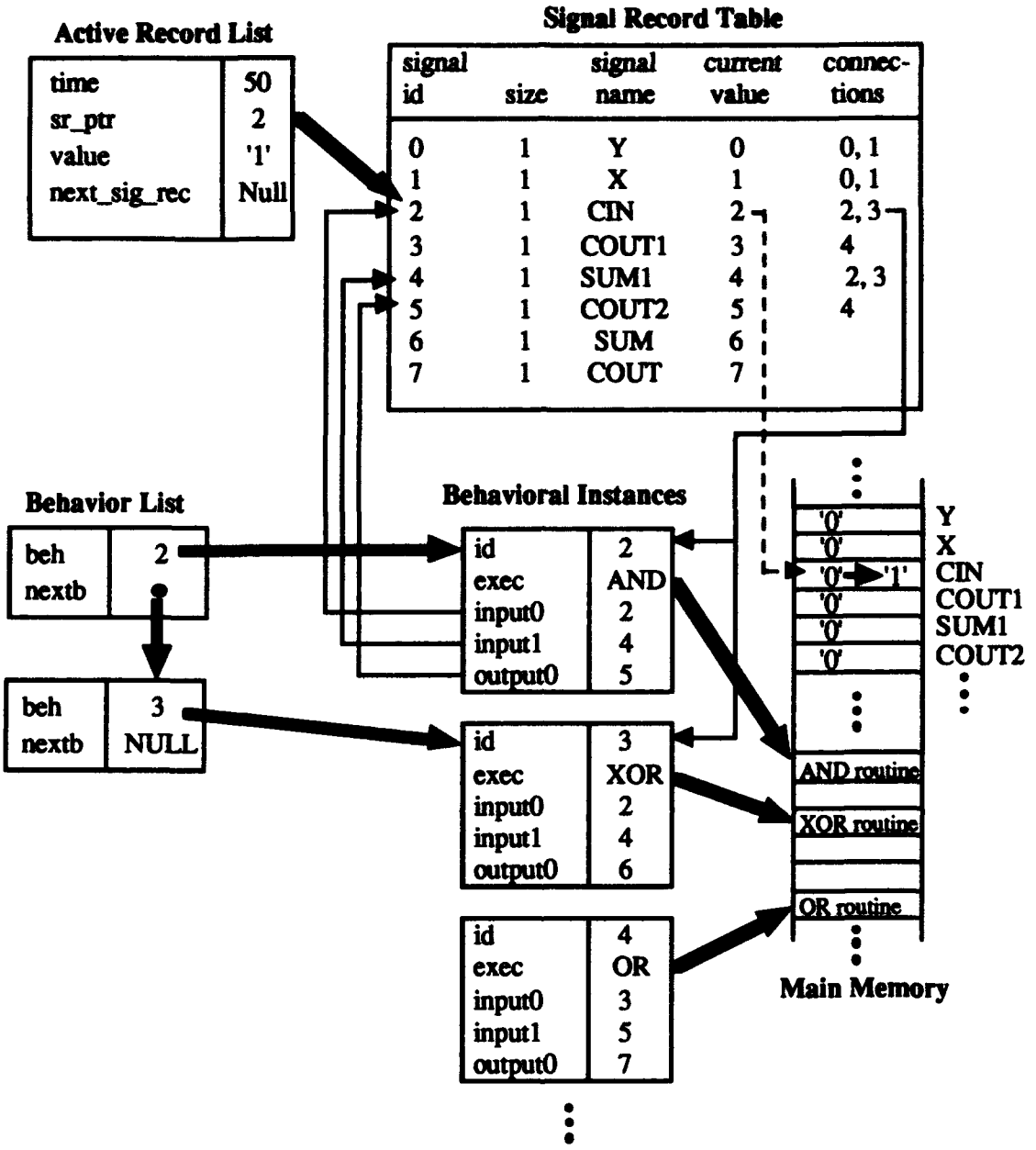


Figure 12. Interrelationship of VHDL Simulation Data Structures (8:3-14)

3.2.1.2 Sequential Simulation Cycle. Figure 13 shows the VHDL simulation cycle used in VSIM's sequential mode. The simulation main loop consists of calls to a series of four specialized routines represented by the circles in the figure. Specifically, the primary simulation routines are (4:25, 8:3-15):

- **Execute Behaviors** - Executes behaviors on the behavior list and removes them from the list. This is where the simulation loop begins.
- **Post** - For each behavior that is executed, this routine posts the corresponding event to the active record list.
- **Get Low Time** - Extracts the entries from the active record list with the lowest next-event time and updates the simulation clock to this lowest time value.
- **Compare Values** - For each entry that is removed from the active record list, this routine compares the new data value in the active record list entry with the current data value stored in memory for that signal. If there is no change in data values, the event is ignored. If there is a change in data values, the affected dependent behaviors are added to the behavior list and scheduled for execution during the next simulation cycle.

As stated previously, all behaviors are added to the behavior list at simulation startup and scheduled for execution at simulation time t = 0. In addition, the current implementation of VSIM requires that all input signal changes be explicitly defined in the VHDL source code. For example, complex processes which randomly generate a new set of input signals during the course of the simulation are not supported by VSIM. As a result of this, all input signal changes (i.e., those specified in the testbench) are added to the active record list at simulation startup (4:26).

The simulation loop shown in Figure 13 continues until both the active record list and the behavior list remain empty for an entire simulation cycle, at which point the simulation is complete (4:26).

3.2.1.3 Handling Behavior Delays. Each behavior execution routine has a nonnegative logical delay, t_{delay}, associated with it. When a behavior has a change on one of its input lines at time t, it is scheduled for execution at time t. When a behavior is

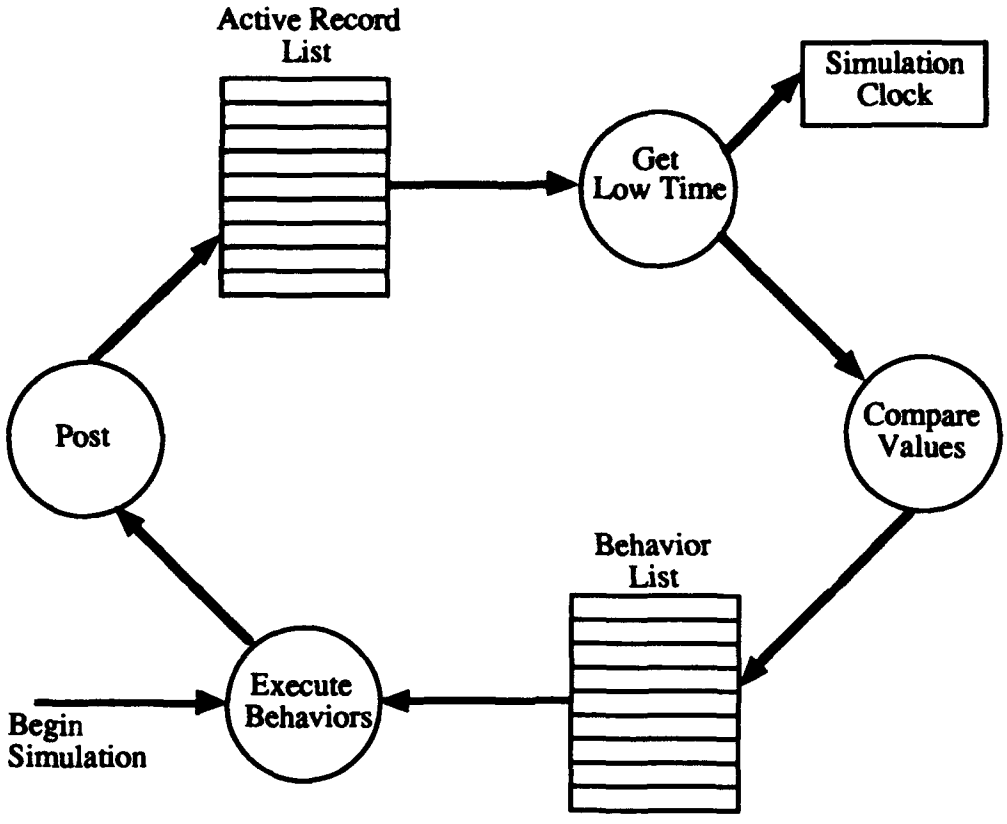


Figure 13. The Sequential VHDL Simulation Cycle (8:3-15)

removed from the behavior list and executed at time t, the resulting output signal is posted to the active record list with a time-stamp of $t + t_{delay}$ (4:27).

If a behavior has n input signals that change value at time t, the behavior will execute n times and the resulting output signal will be posted to the active record list n times. In this situation, the correct event will always be the one corresponding to the most recent behavior execution. As a result, whenever a new event has the same behavior id and time-stamp as another event on the active record list, the new event *replaces* the old event (4:27).

There are two types of delay in VHDL - *transport delay* and *inertial delay*. With transport delay, the output is a function of the input signals regardless of how long the input signals hold their values. With inertial delays, however, the output function requires

that the input signals maintain their values for a time equal to t_{delay} before the output signal may change. This may result in an entry being removed from the active record list if an input signal is not held constant for the required time (4:27). For example, consider the AND gate in Figure 14. A time t = 3 ns, both input signals are '1', and an event for signal Out to go from '0' to '1' at time t = 6ns is added to the active record list based upon the output function of *Out = In1 AND In2 after* t_{delay}. However, at time t = 5 ns, input signal In2 returns to '0' before signal Out has changed values. As a result, the event for signal Out at time t = 6 ns is *removed* from the active record list (4:28).

3.2.2 SPECTRUM Testbed. As shown in Figure 11, VSIM runs over the SPECTRUM parallel simulation testbed. SPECTRUM allows the application to be parallelized by dividing it into logical processes (LPs). The core of SPECTRUM, its "LP manager" (file `lp_man.c`), manages communications between LPs. Function calls to `lp_man.c` are intercepted by a "filter" which provides the specific PDES synchronization protocol being used (4:30). In the case of VSIM, the filter `vhdlclocks.c` implements the

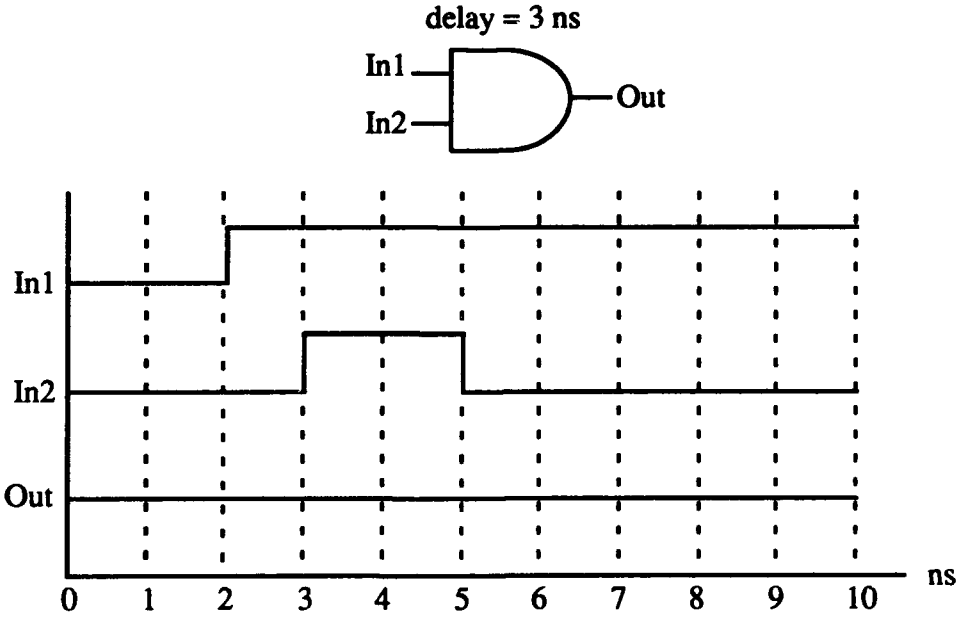


Figure 14. AND Gate with Inertial Delay (4:29)

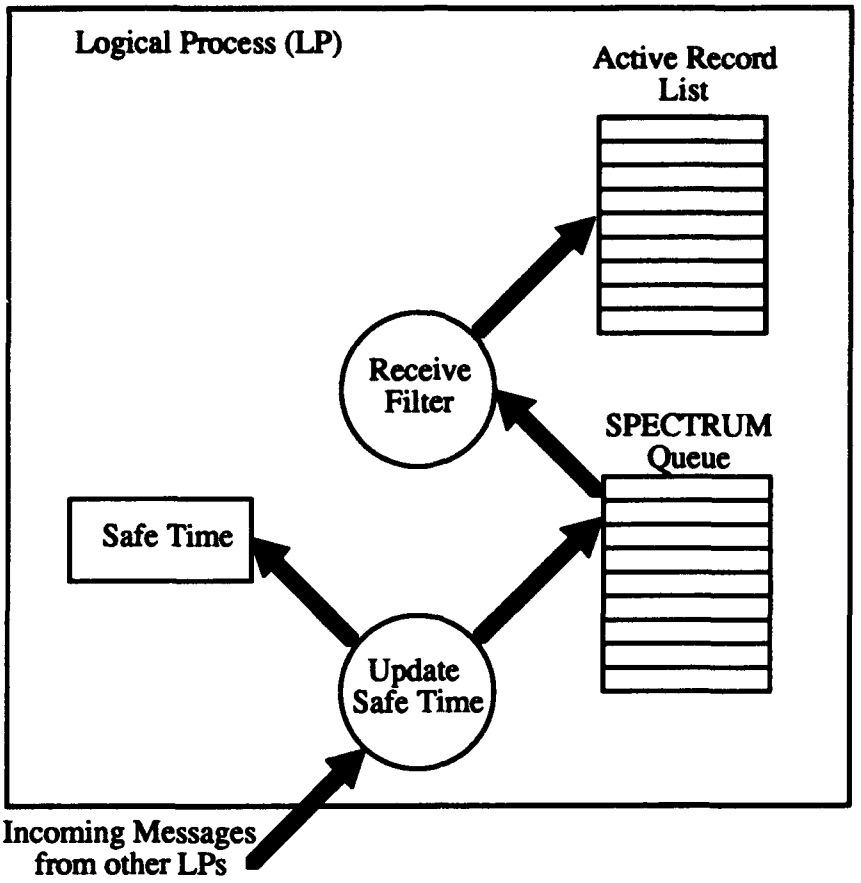


Figure 15. LP Message Receipt (4:35)

conservative Chandy-Misra null message PDES synchronization protocol. Below SPECTRUM's LP manager, the package `cube2.c` provides the interface to the hypercube operating system (4:31). As indicated by Figure 11, each LP contains its own copy of both the application code and the SPECTRUM code. This includes the LP manager, the protocol filter, and the operating system interface.

As shown in Figure 15, SPECTRUM maintains its own queue for incoming messages destined for a given LP. Messages are stored in the SPECTRUM queue until requested by the VSIM application. When VSIM checks for incoming events, messages on the SPECTRUM queue are removed by the "receive filter." The receive filter eliminates synchronization control messages (e.g., null-messages) and places only valid events in the

LP's active record list (4:35). The "safe time" is used to control which events the LP may safely process and will be discussed further in the next section.

The primary functions provided by SPECTRUM's LP manager are (4:30):

- `lp_init()` - Initializes the LP and builds the appropriate filter tables.
- `lp_get_event()` - Retrieves top event from SPECTRUM message queue.
- `lp_post_event()` - Sends event to the owning LP of the affected behavior.
- `lp_advance_time()` - Advances the value of the local simulation clock.
- `lp_terminate()` - Terminates the simulation.

3.2.3 Parallel VSIM Implementation.

3.2.3.1 Parallel Simulation Cycle. When the VHDL behaviors are partitioned among multiple LPs, each LP assumes "ownership" for those LPs in its partition. In VSIM's parallel mode, each LP executes the sequential simulation cycle of Figure 13 for the behaviors it owns. However, because a signal change on one LP may affect behaviors owned by another LP, the simulation cycle must be modified to allow an LP to forward signal changes to other LPs.

Figure 16 shows the simulation cycle modified for parallel simulation. As before, all behavior outputs are posted to the local active record list. When a signal record is removed from the active record list and the value comparison determines that a signal change has occurred, only the affected behaviors owned by the local LP are posted to the behavior list. If the signal change affects behaviors owned by another LP, an event is created and sent to the owning LP where it is eventually placed in that LP's active record list. In this manner, only the results of behavior executions that result in actual signal changes are forwarded to the affected LPs as event messages. Figure 17 shows the parallel simulation cycle for two LPs.

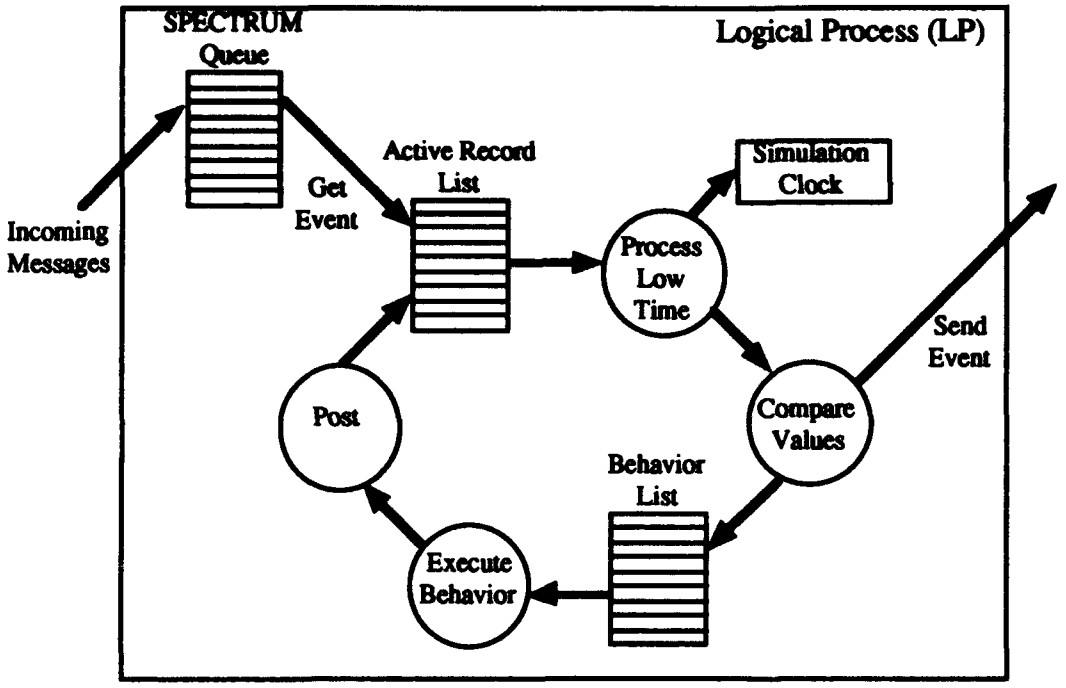


Figure 16. The Parallel VHDL Simulation Cycle for a Single LP (4:34)

3.2.3.2 Synchronization Protocol. VSIM has been designed to execute in a multi-processor, message-passing system with no shared memory. Running over the SPECTRUM testbed, each LP executes an identical copy of the simulation on a disjoint subset of the data (i.e., behaviors). Such a situation is referred to as a single program/multiple data (SPMD) configuration (4:33). In this environment, the parallel simulation cycle of Figure 17 presents an inter-LP synchronization problem caused by dependencies between behaviors owned by different LPs. As an example, consider the hypothetical 2 LP partition for an edge-triggered D flip-flop shown in Figure 18. Each LP maintains a separate simulation clock which maintains that LP's local virtual time (LVT), and indicates how far along in the simulation each LP has proceeded to. In the case of Figure 18, LP1 only has four behaviors to execute as compared to six for LP0. Thus, it is reasonable to expect that LP1 may proceed at a faster rate than LP0. Let t_0 and

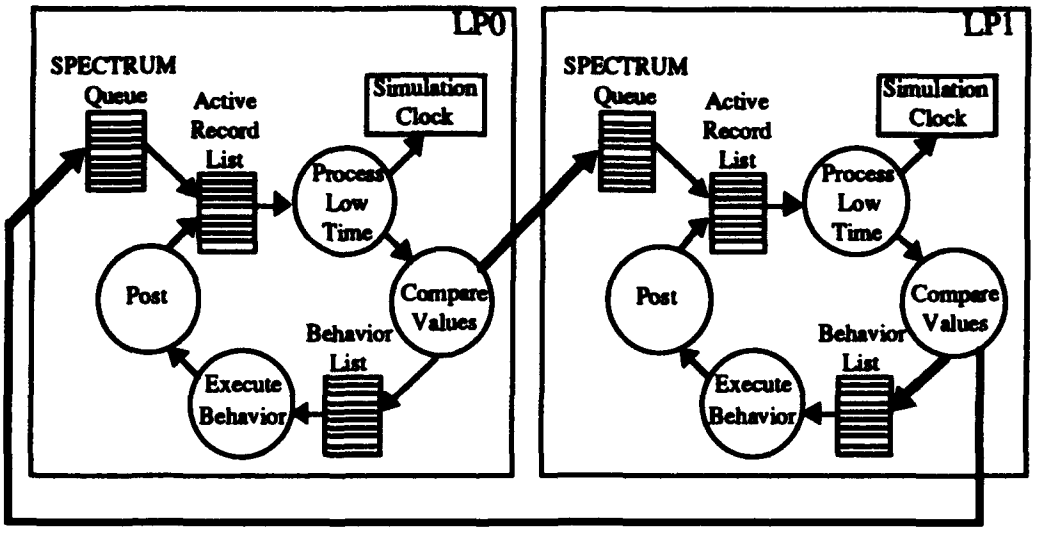


Figure 17. The Parallel VHDL Simulation Cycle for a Two LPs (4:34)

t_1 be the LVT of LP0 and LP1 respectively, such that $t_0 < t_1$. In this situation, it is possible for LP0 to send LP1 an event message with a time that is in LP1's past.

As discussed by Breeden, there are two basic approaches to handling this synchronization problem. In the *optimistic* approach, LP1 would be rolled back in virtual time to a previously saved state at a time equal or prior to the time of the incoming message from LP0. The protocol gets its name from the fact that LP1 would be allowed to proceed as fast as possible with the optimistic assumption that it will not get any late messages from LP0. In the *conservative* approach, LP1 is not allowed to advance its simulation clock to time t_1 unless it is *guaranteed* that it will not receive any messages with a time stamp less than t_1 (4).

Breeden discusses the mechanics of the conservative Chandy-Misra null-message synchronization protocol used in this thesis research (4:11-13, 31-33). Since the null messages add directly to the communications overhead of the partition, the rules for their transmission are reviewed in the next section.

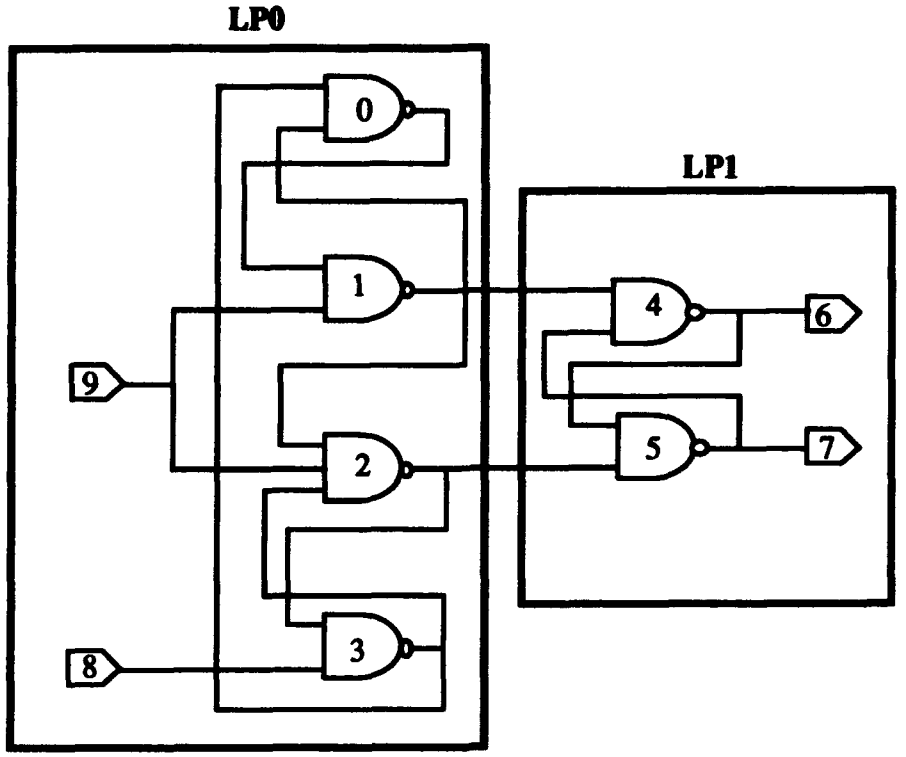


Figure 18. Hypothetical 2 LP Partition for Edge-Triggered D Flip-Flop

3.2.3.3 Null Messages. In order to discuss the rules by which null messages are transmitted, the following variables must be defined (4:32):

- t_{null} - time stamp of the null message.
- t_{neq} - lowest time stamp of all events in an LP's active record list.
- t_{delay} - logical output delay of an LP for a given output line.
- t_{safe} - local virtual time (LVT) that an LP may "safely" approach.

To maintain synchronization, each directed communication link, or line[7], between LPs has a clock associated with it which tracks the time-stamp of the most recent message transmitted over that line. By definition, once a message is transmitted over a line with time-stamp t, it is impossible for a message to be transmitted over that same line with a

[7] The term *line* is used to refer to the directed arcs in the LP conectivity graph. This is not to be confused with the arcs in the problem graph which represent inter-behavior dependencies. Given N LPs, each LP can have at most (N-1) output lines, but each output line may consist of multiple inter-behavior arcs.

time-stamp less than t. The safe-time, t_{safe}, is the minimum time of all input lines, and represents the maximum time to which the LP may approach with the guarantee that no messages will be received with earlier time-stamps. As shown in Figure 3.15, the safe-time is updated each time an LP receives a message from another LP. This is accomplished by comparing the updated time-stamps of each input line to find the minimum. Null messages are used to advance the times associated with each line (thus advancing t_{safe}), and otherwise contain no useful information (4:32).

As implemented in the filter `vhdlclocks.c`, null messages are sent under the following circumstances (4:32-33):

- By definition, an LP is constrained to process events in non-decreasing order of simulation timestamps. As such, when an LP processes an event at time t, it is guaranteed that it will not process an event in the future with a timestamp of less than t. Therefore, when LP_n sends an event message over one of its output lines with a time-stamp of t, it sends a null message over all other output lines with a time-stamp of t to let all other LPs know that they will not receive an event message from LP_n with a timestamp less than t. This allows them to advance the clock associated with the input line from LP_n to time t.
- When there are no event messages waiting in the SPECTRUM input queue at the time of a request by VSIM, the receive filter (see Figure 15) checks to see if there is an event on the local active record list that has a time-stamp less than or equal to the safe time (i.e., $t_{neq} \leq t_{safe}$). If so, a null pointer is returned to VSIM and processing may continue. If not, the filter blocks, waiting for an incoming message to advance the safe time. Prior to blocking, however, the LP sends a null message over each of its output lines with a time-stamp equal to the smaller of the safe time plus the LP delay for that output line, or the time at the top of the local active record list (i.e., $t_{null} = min\ ((t_{safe} + t_{delay}), t_{neq})$). These null messages

serve as guarantees to the receiving LPs that no events prior to t_{null} will be received from the sending LP. This allows the receiving LPs to update their safe times, thus avoiding cyclical waiting and preventing deadlock.

In Breeden's version of this protocol, null messages were also sent over each LP output line at simulation startup with a timestamp of that output line's delay value. The purpose of these null messages was to advance the safe time of each LP beyond zero so that each LP may begin processing events. These null messages have been eliminated since all behaviors in the system are automatically scheduled for execution at simulation startup.

3.2.4 Code Transformation. Circuit specific information is contained in the intermediate C code created during the model generate phase of compilation as shown in Figure 10. This circuit specific code includes routines to instantiate each behavior and signal and to execute the behavioral output functions. Breeden defines a seven step process by which this intermediate C code is transformed into code compatible with VSIM (4:29-30).

3.3 Partitioning Requirements

3.3.1 Load Balancing. Load balancing is defined as the degree to which all available processors are assigned an equal share of the computation load (26:59). In parallel VHDL simulations, the *computation load* consists of processing signal changes, scheduling the affected behaviors, and executing the affected behaviors (which in turn may cause further signal changes). One method of measuring load imbalance is to calculate the difference in the minimum and maximum finishing times of all processors as a percentage of the maximum finishing time (26:59):

$$H_b = \frac{t_{max} - t_{min}}{t_{max}}$$

However, this measure of load imbalance is only valid if all processors are able to perform their share of the workload completely independent of the others. In most parallel simulations, there exist dependencies between the workloads that have been assigned to different processors. For example, consider the example VHDL simulation of Figure 19 in which an edge-triggered D flip-flop has been partitioned into three LPs. LP0 has ownership of eight behaviors, versus one each for LP1 and LP2. Intuitively, LP0 will process a majority of the signal changes and behavior execution routines. Thus, LP0 has been assigned a clear majority of the computation workload. However, because of inter-behavior dependencies which cross partition boundaries, LP1 and LP2 cannot complete their share of the computation load until they receive the last event message from LP0. As a result, while LP0 is processing a large number of events, LP1 and LP2 spend time blocking for inputs from LP0. All three LPs will finish at approximately the same time (with LP0 finishing slightly ahead of the other two), even though LP0 had a much greater share of the computation load.

For parallel VHDL simulation, an alternative method of measuring load imbalance is required. The method proposed in this thesis uses the following definitions:

b_i = behavior i.

w_i = relative computation cost of behavior i.

B_q = set of all behaviors assigned to LP q.

L_q = the total computation load of LP q.

L_{max} = the maximum computation load of all LPs.

L_{avg} = the average computation load of all LPs.

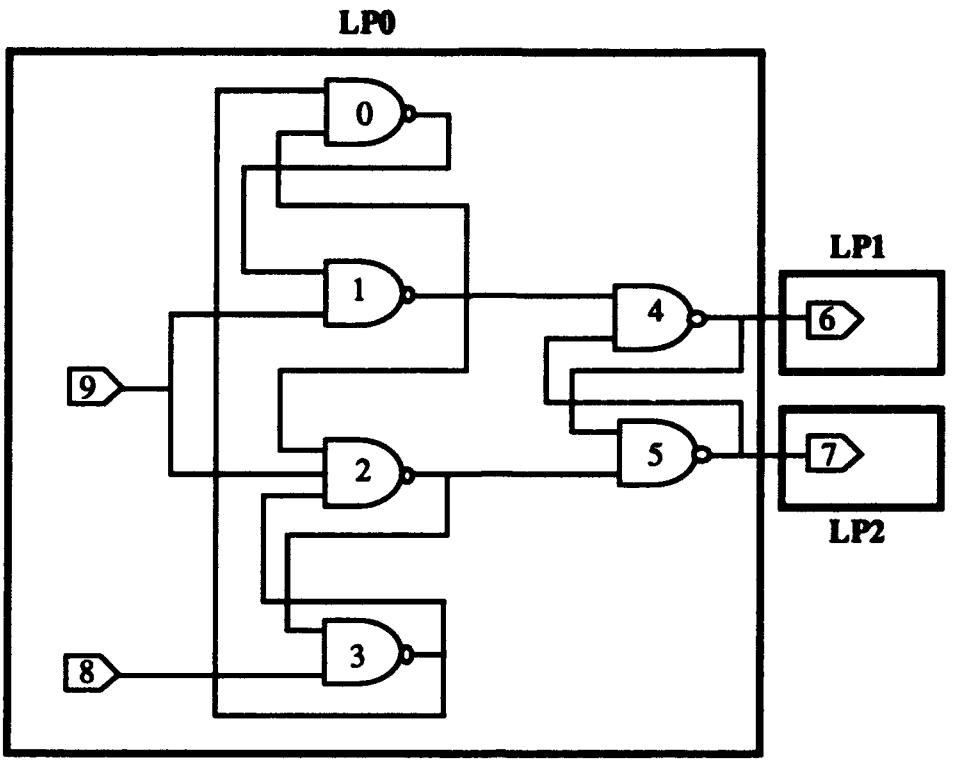


Figure 19. Load Imbalance Example

With these variables, the computation load for each LP can be calculated as:

$$L_q = \sum_{b_i \in B_q} b_i w_i$$

And the load imbalance factor, H_b, can be calculated as:

$$H_b = \frac{L_{max} - L_{avg}}{L_{avg}}$$

The relative computation cost of each behavior, w_i, is actually composed of two components: the relative computational complexity of behavior i, and the relative frequency that behavior i is executed during the simulation. In VSIM, all behaviors are simple VHDL processes.[8] Intuitively, the major difference in the computational intensity of these simple VHDL processes is in the number of input signals to be evaluated. If the number of behavior inputs is kept relatively small (e.g. ≤ 4), these computational

[8] Simple boolean operation (AND, OR, NOT, assignment, etc.) with a finite number of inputs.

differences will be negligible when compared to the computation requirements of more complex VHDL processes (such as a bus resolution function with 32 inputs). As such, the relative computational complexity of all behaviors representing simple VHDL processes can be safely assumed to be equal.

The relative frequency with which a behavior is executed is heavily dependent upon the *activity*[9] of its input signals during the simulation (23:85). Experience has shown that the signal activity of a typical VHDL circuit is not evenly distributed. However, prior to simulation, no data is available regarding signal activities (23:85). Since no information is available to support a specific behavior weighting, all behaviors are assumed to be equally affected by the circuit signal activity.

Since behaviors cannot be differentiated in terms of computational complexity or execution frequency, all behaviors are evenly weighted ($w_i = 1$), and all mapping decisions must be made solely on the static inter-dependency structure of the VHDL circuit (23:85). Therefore, the computation load L_q is simply calculated as the number of behaviors assigned to LP q.

3.3.2 Minimizing Communications Costs. Inter-process communications can be defined as the sending of information from a source process to a destination process with the destination process requiring the sent information in order to progress. In VSIM, each behavior is a simple VHDL process, and the sending of a signal change from the output of one behavior to the input of another behavior is one form of inter-process communications. In VSIM's sequential mode, this form of inter-process communications consists of inserting a record in the behavior list as shown in Figure 13. In VSIM's parallel mode, this form of communications is identical *if* the receiving behavior is owned by the same LP as the sending behavior. However, if the sending and receiving

[9] The *activity* of a signal is defined as the number of events generated for that signal during the simulation (23:85).

behaviors are owned by *different* LPs, the signal must be sent as an inter-LP *event message* using the SPECTRUM layer as shown in Figure 17. The insertion of the record into the behavior list occurs at the receiving LP after the event message has been removed from the receiving LP's SPECTRUM incoming message queue. Thus, the sending and receiving of inter-LP event messages by SPECTRUM represents a communications overhead not present in the sequential simulation.

In parallel VSIM, communications between behaviors that are owned by the same LP do not represent additional communications overhead over the sequential version, and are said to have a cost of zero. Rather, the communications overhead that we are interested in minimizing is the inter-LP event message traffic that occurs when communicating behaviors are owned by different LPs. Throughout the remainder of this thesis, the term *inter-process* communications will be used to describe this message level communications between the logical processes. With the previous assumption of one LP per physical processor, this inter-process communications is also equivalent to the *inter-processor* communications, and the two terms can be used inter-changeably to refer to the communications overhead of the parallel simulation.

3.3.2.1 Modeling Inter-Process Communications. The model used to represent inter-process communications used in this thesis makes use of a *Communications Weight Matrix*, and the following definitions:

w_{ij} = the relative cost of communications between processor i and processor j based upon the topological layout of the processors.

a_{ij} = the number of directed inter-behavior dependencies (i.e. arcs) between LP_i and LP_j.

With these definitions, the total cost of directed communications from LP_i to LP_j, C_{ij}, can be calculated as:

$$C_{ij} = a_{ij} w_{ij}$$

Ideally, each arc that makes up the factor a_{ij} would be multiplied by a factor that accounts for the frequency of the communications over that arc. However, the frequency of communications is dependent upon the signal activity, and, as stated in section 3.3.1, there is no information available regarding signal activity prior to simulation (23:85). Therefore, the communications costs must be estimated based upon the known structural dependencies that cross the LP boundaries. In addition, the assumption that LP0 will be mapped to processor 0, LP1 will be mapped to processor 1, etc., is implicit in the equation for C_{ij}.

Letting n be the number of logical processes (LPs), the inter-process communications can be represented by an n x n communications weight matrix as shown in Figure 20. The main diagonal entries represent the cost of an LP's communications with itself. As stated previously, these communications do not involve the sending and receiving of event messages, and thus do not contribute to the parallel simulation communications overhead. Therefore all main diagonal entries are always 0.

Letting H_c represent the overall total inter-process communications costs, it can be calculated as the sum of the entries in the communications weight matrix:

$$H_c = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} C_{ij}$$

With the exception of a constant factor of 1/2, this equation is identical to the equation used by Bultan to calculate the communications cost sub-function H_c in the mean field annealing algorithm discussed in section 2.4.11 (5:296).

Upon further examination, however, a potential problem arises because H_c does not give an accurate picture of the total inter-process communications relative to the total signal change activity of the circuit (as estimated by the number of inter-behavior arcs). For example, if circuit A has a total of 1,000 inter-behavior arcs with 100 crossing LP

$$\begin{bmatrix} 0 & C_{01} & C_{02} & C_{03} & \cdots & C_{0(n-1)} \\ C_{10} & 0 & C_{12} & C_{13} & \cdots & C_{1(n-1)} \\ C_{20} & C_{21} & 0 & C_{23} & \cdots & C_{2(n-1)} \\ C_{30} & C_{31} & C_{32} & 0 & \cdots & C_{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & & \vdots \\ C_{(n-1)0} & C_{(n-1)1} & C_{(n-1)2} & C_{(n-1)3} & \cdots & 0 \end{bmatrix}$$

Figure 20. Communications Weight Matrix for n LPs

boundaries and circuit B has a total of 10,000 inter-behavior arcs with 500 crossing LP boundaries, circuit B will have a higher H_c than circuit A even though it has a smaller percentage of its arcs crossing the LP boundaries. To account for this, it is desirable to calculate the total inter-process communications costs as a percentage of the total communications costs in the system. Thus, the equation for H_c is modified as follows:

$$H_c = \frac{\sum_{i=0}^{n-1} \sum_{j=0}^{n-1} C_{ij}}{\text{num\_arcs}}$$

where num_arcs is the total number of inter-behavior arcs in the system. Minimizing the value of H_c is one of the primary goals of the partitioning algorithms implemented for this thesis.

3.3.2.2 Distribution of Communications. In addition to the *amount* of inter-process communications, the *distribution* of those communications may also add to the overhead of the parallel simulation. For example, consider the four LP examples of Figure 21 in which it is assumed that all LPs are assigned an equal share of the computation workload. In Figure 21.a, LP2 and LP3 run independently of all other LPs, while there is a relative communications cost factor of 100 from LP0 to LP1. As a result, LP2 and LP3 will finish in minimal time while LP1 cannot finish ahead of LP0 because

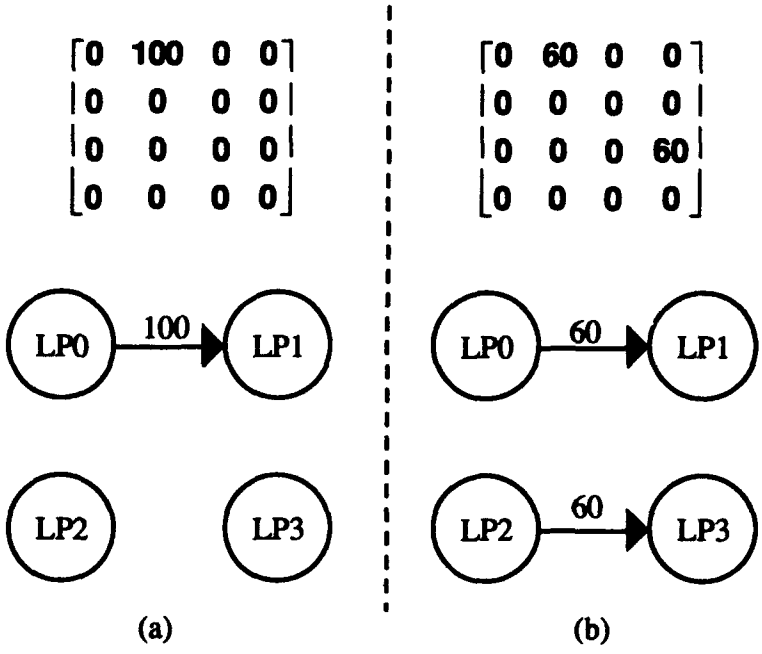


Figure 21. Communications Distribution Example

of the inter-LP dependencies. In this case, the communication costs between LP0 and LP1 form a *bottleneck* which holds up overall simulation completion[10]. In Figure 21.b, the problem has been repartitioned in order to split up the communications bottleneck between LP0 and LP1 at the expense of a higher total communications cost. Although the overall communications costs are higher, the simulation should reach completion faster because of the reduced bottleneck in the communications costs.

The results presented in Chapter 5 show that in some circumstances, it is possible to partition a circuit such that the total amount of inter-process communications is reduced by more than 33%, but the resulting simulation performance is worsened with strong evidence that this is due to a bottleneck in the distribution of the remaining inter-process communications. As a result, the calculations for the total communications cost overhead in this thesis include an additional factor in order to account for the distribution of the inter-LP dependencies in the circuit partition.

[10] Simulation completion is defined as the completion time of the *slowest* logical process.

The calculation of the communications distribution factor H_d uses the following assumptions and definitions:

- Message setup and transmission time is much more significant than the time it takes to receive an incoming message. This assumption is based upon an analysis comparing the overhead involved in sending vs. receiving an inter-processor message. A message receive action involves inserting the message in a queue where it is held until actually needed by the simulation. On the other hand, a message send action involves the message setup and transfer times, as well as blocking time while the sending processor waits for a free communications link. Intuitively, the sending processor has a greater level of overhead. Instrumentation of the simulation is required in order to fully validate this assumption.
- An LP's contribution to the total communications cost overhead is directly proportional to the total weighted communications costs associated with all arcs leaving that LP. This is represented by the sum of the row in the communications weight matrix corresponding to that LP.

D_{avg} = the average weighted communications costs associated with each LP.

D_{max} = the maximum weighted communications costs associated with an LP.

The communications distribution factor is then calculated as the maximum positive deviation from the average as a percentage of the average:

$$H_d = \frac{D_{max} - D_{avg}}{D_{avg}}$$

3.3.2.3 Effect of Lookahead. Simulation *lookahead* is defined as the ability to predict what will happen or not happen in the future with complete certainty. A process at simulation time t with lookahead t_L will be able to accurately predict all events it will generate up to simulation time $(t + t_L)$ (11:9). In the case of VSIM, an LP's lookahead

refers to the ability to predict a simulation time in the future up to which that LP can *guarantee* that it will generate no signal changes destined for other LPs. The LP's lookahead is defined by *min* $((t_{safe} + t_{delay}), t_{neq})$.

As discussed in section 3.2.3.3, the value t_{delay} is defined as the minimum output delay associated with each of the LP's output lines. These delay values are specified in a ".arcs" file that is read in by VSIM at runtime. In previous research, a uniform LP delay time was used that was equal to the smallest non-zero delay in the circuit associated with a single behavior (4). In order to guarantee optimal LP delay values, however, this thesis uses the minimum path from all possible LP input lines and all source behaviors in the LP to each LP output line in calculating the corresponding output line's delay value. For a random partition, this method generally results in no net improvement in the lookahead of the circuit. However, for more sophisticated partitioning algorithms, .arcs files with larger LP delay values can be obtained. When $(t_{safe} + t_{delay}) < t_{neq}$, a larger delay value will result in a null message with a higher time stamp being sent. In turn, this will result in fewer null messages being sent over that output line, and *may* allow the safe time of the receiving LP to advance at a faster rate. More discussion of the effect of increasing the lookahead of the circuit is presented in Chapter 5.

3.3.2.4 Null Messages. In VSIM, the conservative Chandy-Misra synchronization protocol adds additional communications overhead in the form of *null messages* which transmit no useful information[11]. As discussed in section 3.2.3.3, null messages are sent according to the given set of rules in order to avoid deadlock by updating the safetimes of the receiving LP.

Analysis of the rules for sending null messages shows that the number of null messages transmitted is primarily dependent upon the number of arcs in the LP inter-

[11] As opposed to *real messages* which transmit actual signal value information.

connectivity graph (i.e. the number of output lines as specified in the associated .arcs file). This is also equivalent to the number of non-zero entries in the communications weight matrix. Other factors which also have an effect on the number of null messages sent are the minimum LP delay values specified in the .arcs file, and the number of inter-behavior arcs which cross LP boundaries.

The simulation results presented in Chapter 5 show that the ratio of null messages to real messages in a simulation grows with the number of processors, with null messages dominating the communications costs for partitions with more than 2-4 LPs for the circuits used in this thesis. For example, for the wallace tree simulation with a random partition, the null message to real message ratio goes from approximately 1:4 with 2 LPs to approximately 5:1 with 8 LPs. The equations in sections 3.3.2.1 and 3.3.2.2 for H_c and H_d do not directly account for the transmission of the null messages required to maintain simulation synchronization. Therefore, an additional cost factor is needed to take into account this sizable overhead.

Letting L_{arcs} be the amount of lookahead in the .arcs file, and O_{arcs} be the number of LP output lines specified in the .arcs file, the null message communications factor can be defined as:

$$H_n = L_{arcs} O_{arcs}$$

The dependence of the number of null messages on the number of inter-behavior dependencies that cross LP boundaries is included in H_c, and is not included again in H_n.

Given N LPs, each LP can be connected to at most N-1 LPs. Thus, the limit on the value of O_{arcs} is given by:

$$O_{arcs} \leq N(N-1)$$

The value of L_{arcs} is normalized to 1.0 for the case when all LP delays in the .arcs file are equal to the smallest non-zero behavior delay in the system. Letting the smallest

non-zero behavior delay in the system be called the normal lookahead, the value of L_{arcs} can be calculated as:

$$L_{arcs} = \frac{\text{normal lookahead}}{\text{actual average lookahead}}$$

If the actual average lookahead is made to be smaller than the normal value, L_{arcs} will be greater than 1.0, thus compounding the impact of O_{arcs} on the total communications costs. On the other hand, if the actual average lookahead is larger than the normal value, L_{arcs} will be smaller than 1.0 and the effect of O_{arcs} will be abated.

It should be noted that because the LP delay is not always used in computing the timestamp of a null message (i.e. when $t_{neq} < t_{safe} + t_{delay}$), the value of L_{arcs} will only provide an estimate of the effect of increased lookahead on the null message overhead. For this reason, a better estimate of the impact of increased lookahead can be calculated by making a reasonable assumption regarding the percentage of time that the delay value determines the timestamp of a null message. The assumption used in this thesis is 50%, resulting in the lookahead increase being cut in half. It should be noted that the uneven distribution of signal activity in the circuit will affect the accuracy of this assumption. However, chapter 5 includes data on the effect of increased lookahead which shows this assumption to provide good estimates under most circumstances The equation of L_{arcs} is adjusted as follows:

$$L_{arcs} = \frac{\text{norm\_lookahead}}{\left(\frac{\text{norm\_lookahead - avg\_lookahead}}{2}\right) + \text{avg\_lookahead}}$$

3.3.3 Balancing Load Imbalance and Communications Costs. Intuitively, there is a natural conflict between the partitioning goals of minimizing the inter-process communications costs and assigning each LP an equal share of the computation load. For example, by assigning all 10 behaviors in Figure 18 to LP0, the inter-process

communications costs can be eliminated completely. However, the problem is reduced to a sequential simulation with LP0 carrying the entire computation load while LP1 sits idle. On the other hand, making the partition as shown in Figure 18, the simulation is closer to being balanced in terms of computation load at the price of adding inter-process communications costs.

The primary goal of any partitioning algorithm is to strike a balance between these two conflicting goals that results in a good simulation performance. The exact nature of this balance, however, is dependent upon the relative performance of the CPU and communications subsystem of the hardware platform being used.

3.3.4 Measuring the Cost of a Partition. Measurement of the simulation performance resulting from a given partition is the ultimate method of determining the quality of a partition. However, many partitioning algorithms, including the one implemented in this research, require an interim assessment of the quality of the partition at various points in the algorithm. To achieve this interim assessment, an objective cost function is used similar to the one used for the mean field annealing algorithm described in section 2.4.11.

3.3.4.1 Objective Cost Function. The objective cost function is composed of the factors for load imbalance and communications costs discussed in sections 3.3.1 and 3.3.2. Specifically, the objective cost function H is defined as:

$$H = \beta H_n H_c (1 + H_d) + \alpha H_b$$

where α and ß are constant coefficients which control the relative influence of the communications and load balancing portions of the equation.

3.3.4.2 Relationship to Simulation Performance. One of the objectives of this research was to quantify the relationship between the quality of a partition and the resulting performance of the simulation. The performance of the simulation is measured in terms of *speedup*, which is defined as the ratio of the execution time on a single processor to the execution time on P processors (26:65):

$$Speedup = S_p = \frac{t_{one\ processor}}{t_{P\ processors}}$$

Ideally, the time to execute on P processors will be 1/Pth the time to execute on a single processor, giving a speedup of P:

$$S_p = \frac{t_{one\ processor}}{\left(\frac{t_{one\ processor}}{P}\right)} = \frac{P}{1} = P$$

In general, however, the simulation overhead will prevent the achievement of a speedup of P on P processors. As discussed previously, the simulation overhead is directly related to the quality of the simulation partition as measured by the objective cost function H. It is possible to estimate the expected simulation speedup by relating the cost function H to the speedup S_P as follows:

$$S_p = \frac{P}{1 + \gamma H} = \frac{P}{1 + \gamma\left[\beta H_n H_c (1 + H_d) + \alpha H_b\right]}$$

where γ is a constant coefficient. Note that in a perfect partition (i.e. no inter-processor communications and equally balanced loads), the cost function will equal 0 and the speedup will be equal to the number of processors. In addition, note that if $(1 + \gamma H)$ increases at a faster rate than P, then the estimated speedup will decrease as the number of processors is increased.

3.4 Partitioning Approach

The primary partitioning algorithm implemented in this thesis, referred to as ***AB-Annealing***, draws upon several properties of the partitioning strategies presented in section 2.4. AB-Annealing is a multi-phased partitioning strategy with the following steps:

- The graph is partitioned into strong-components.
- Treating the strong components as indivisible blocks, the graph is divided into the required number of LPs using a deterministic graph-traversal algorithm.
- Given the initial partition from the previous step, those behaviors that lie on the boundary between two LPs are considered for reassignment to a different LP on a priority basis based upon the potential reduction in the objective cost function.

The phased approach to the mapping problem used here is similar to the phased approaches used in the Two-Dimensional Algorithm of section 2.4.5 (22) and Algorithm H of section 2.4.7 (16). The initial partition step uses a simple graph-traversal algorithm which has many similarities with the Depth-First Breadth-Next algorithm of section 2.4.8 (17). The final step implements a "border-annealing" algorithm in an attempt to iteratively improve the partition by making behavior reassignments that result in a decrease in the objective cost function. The objective cost function and iterative nature of this phase make it similar to a deterministic version of the Mean Field Annealing algorithm of section 2.4.11 (5). However, the fact that only those behaviors that lie on the border between two LPs are considered for reassignment also makes this phase similar to the Two-Dimensional Algorithm of section 2.4.5 (22).

3.4.1 Strong Components. The first step of the AB-Annealing process involves finding the strongly connected components of the problem graph. Each strong component in the problem graph corresponds to a complete feedback loop in the VHDL circuit. This

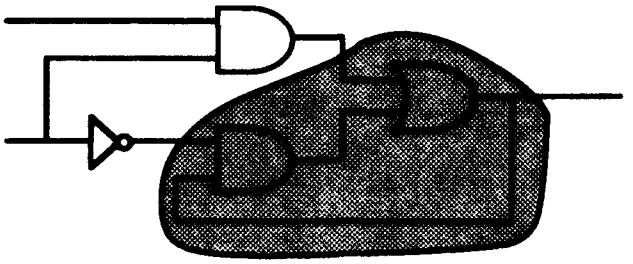


Figure 22. Strong Component Example - Simple Latch Feedback Loop

step is included in order to isolate the feedback loops in a single LP during the initial partition. An example strong component common in digital circuits is shown in Figure 22. The problem graph in the example represents a simple latch with a two-behavior feedback loop.

3.4.2 Initial Partition. In the Mean Field Annealing algorithm, the initial partition is determined with a random function. In the AB-Annealing algorithm, it was desired to use the available knowledge about the structure of the circuit in an effort to make a good first-cut partition. It was anticipated that this approach, in addition to being deterministic, would reduce the amount of work necessary during the annealing phase as well as lead to a better final partition. Two variations on the DFBN algorithm of section 2.4.8 have been implemented to serve as the initial partitioning routines for the AB-Annealing algorithm.

In the first algorithm, referred to as Simple Depth-First (SDF) partitioning, the problem graph is traversed in a depth-first manner. When a behavior is visited for the first time, it is added to the current partition and marked so that it will not be added to any other partitions. Before beginning a partition, its size is pre-determined using the number of unmarked behaviors divided by the number of unfilled partitions. If a newly visited behavior is part of a strong component, the entire strong component is added to the current partition. When the current partition is full, a new partition is started.

The second algorithm, referred to as Simple Breadth-First (SBF) partitioning, the problem graph is traversed in a breadth-first manner, but the partitions are built in the same way as with the SDF partitioning.

For comparison purposes, the random partitioning algorithm can also be used to generate the initial partition for the AB-Annealing algorithm. In the random case, however, the first step of finding the strong components has no meaning and is omitted.

3.4.3 Border-Annealing. The third and final phase of the AB-Annealing algorithm consists of an iterative improvement of the initial partition through selective reassignment of certain behaviors that lie on the border of the partition. This phase is referred to as "border-annealing," and involves the following steps:

- Calculate the priority of each behavior based upon the following formula:

  Priority = Max_External_Arcs - Local_Arcs

  where Local_Arcs represents the number of input and output arcs of the given behavior that are to or from behaviors in the same partition, and Max_External_Arcs is the maximum number of input and output arcs of the given behavior that are to or from behaviors in any single partition other than the given partition.
- Place each behavior with a priority ≥ 0 in an annealing queue in decreasing order of priority. By definition, this eliminates from consideration all behaviors that are not on the border of a partition (such behaviors will have a priority < 0).
- Remove each behavior from the queue in priority order and evaluate the effect of all possible moves on the objective cost function.
- • Based upon the data produced in the previous step, select the best move that will not cause the load delta factor to become larger than the maximum load imbalance tolerance (user defined input parameter). It is possible that no move may be selected.

- • If a move is selected, carry out the move and update the data structures used in the calculation of the objective cost function.
- Repeat the above steps until the maximum number of iterations (user defined input parameter) has been reached, or until no more improvement can be made.

3.4.3.1 Selecting Moves for Consideration. In the AB-Annealing algorithm, behaviors are selected for potential LP reassignment based upon their priority. Using the behavior prioritization scheme discussed above, behaviors that have no input or output arcs which cross an LP boundary will not be considered for moving. In addition, behaviors which have more input and output arcs that stay within its own LP than go to any single external LP will also be eliminated from consideration.

Figure 23 shows several examples of behavior priorities. In Figure 23.a, behavior 3 in LP0 has two arcs connected to behaviors in LP1, but only one arc connected to a behavior in its own LP. Thus, behavior 3 has a priority of +1, and will be queued up for move consideration. In Figure 23.b, the two external arcs of behavior 3 are connected to two different LPs. The maximum number of external arcs to a single LP is 1, and the priority of behavior 3 is 0. In this situation, a move can be made to improve load balancing, or some other factor, without any net change in the number of inter-LP arcs. In Figure 23.c, behavior 3 has more arcs connected to behaviors in its own LP than any other LP. Thus, its priority is negative, and it will not be placed in the annealing queue for move consideration.

Figure 24 shows the actual SDF initial partition for the edge-triggered D flip-flop circuit. The behaviors that lie on the boundaries of the partitions are highlighted. The priorities for each behavior are listed in Table 1. Note that in this example, only behavior 8 has a nonnegative priority. Thus, only behavior 8 will be considered for LP reassignment.

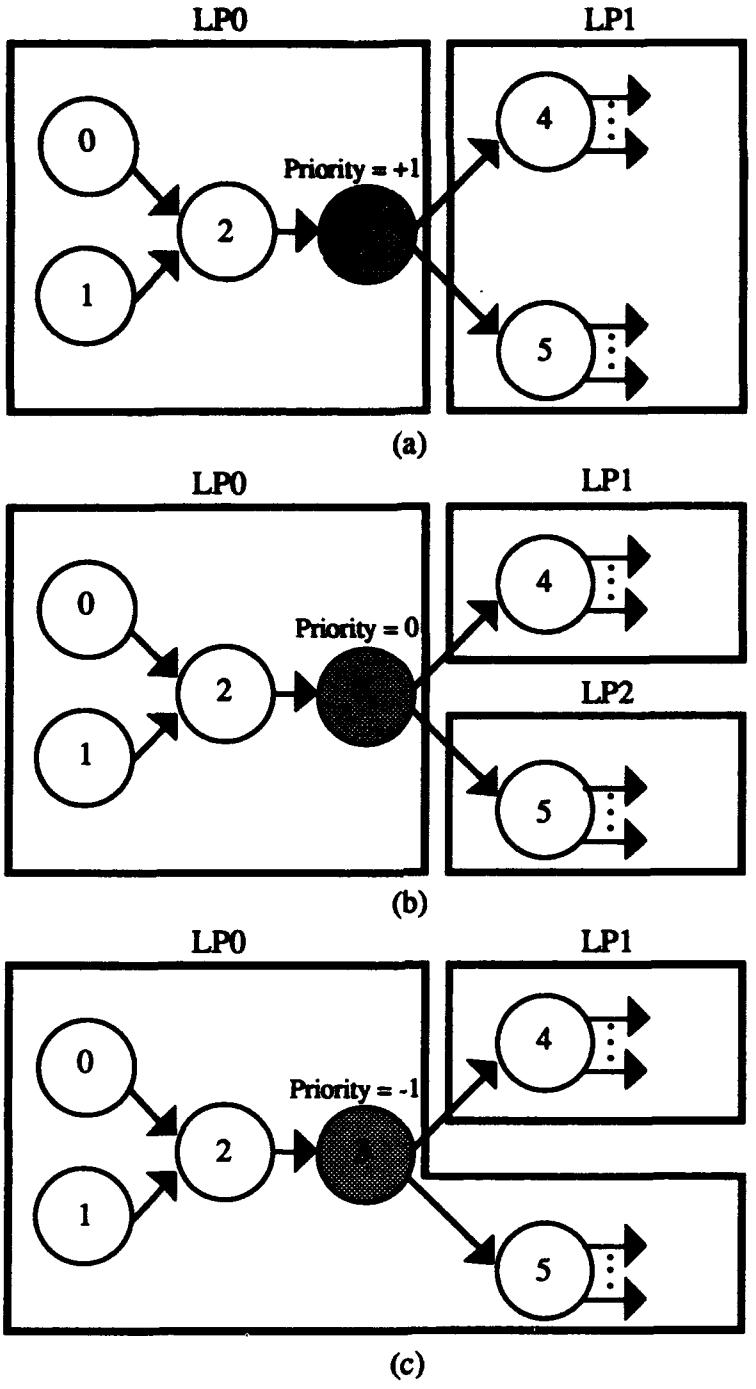


Figure 23. Example Behavior Annealing Priorities

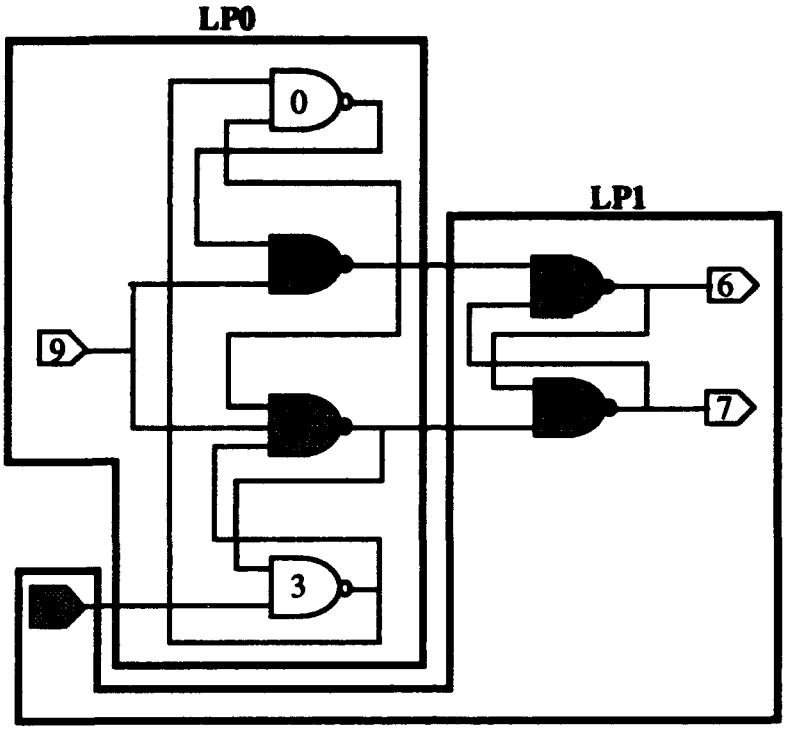


Figure 24. SDF Initial Partition for Edge-Triggered D Flip-Flop

Table 1. Behavior Priorities for the Partition of Figure 24

Behavior	Max_External_Arcs	Local_Arcs	Priority
0	0	3	-3
1	1	4	-3
2	1	4	-3
3	1	3	-2
4	1	3	-2
5	1	3	-2
6	0	1	-1
7	0	1	-1
8	1	0	+1
9	0	2	-2

Because a selected move may effect the priority of other behaviors in the annealing queue, it is not possible to select more than one move at a time. A behavior can only be fully evaluated regarding all prospective moves when it reaches the top of the annealing queue. The idea behind using the annealing queue is to narrow down the set of behaviors

that must be evaluated to only those that have a good chance of qualifying for a move. Under this approach, it is possible that a selected move will cause the priority of a non-queued behavior to become nonnegative. In this situation, the affected behavior would be queued during the next iteration. The alternative to this approach would be to iterate through all of the behaviors after each move, selecting the single best candidate behavior. This approach has two disadvantages. First, it is computationally more expensive, requiring $O(NP^2)$ worst case per potential move, vs. $O(P^2)$ (note that N is the number of behaviors and P is the number of logical processes). Second, a complex scheme of tracking which behaviors have been considered for reassignment must be implemented along with a prioritization mechanism to prevent the starvation of low priority behaviors.

3.4.3.2 Solution Convergence. The iterative border-annealing process described above continues until one of two things occurs:

- The algorithm converges to a solution in which no more progress can be made. This is indicated by an entire iteration in which no moves are selected.
- The maximum number of iterations is reached.
- A specified number of consecutive iterations are processed with no net improvement in the objective cost function.

To allow for the evaluation of slight variations in the annealing process, three values are taken as modifiable input parameters to the annealing algorithm:

- `Num_Iterations` - defines the maximum number of annealing iterations to perform before terminating the process.
- `Max_Worthless_Iterations` - defines the number of consecutive iterations with no net improvement in the objective cost function which can be processed before the annealing process is terminated.

- **`Load_Imbal_Tol`** - defines the maximum value of the load delta factor that is acceptable. Moves which cause the load delta factor to be larger than the `Load_Imbal_Tol` will not be made, even if they lower the overall inter-process communications costs.

3.4.4 Topological Variation. As discussed in section 2.2.1, topological variation arises when the inter-dependency structure of the parallel application differs from the inter-connectivity structure of the parallel system (2). In the case of VSIM, grouping the circuit behaviors into logical processes (LPs) does not eliminate the problem of topological variation, as the inter-connections between the LPs may still differ from the inter-connectivity structure of the hypercube.

Figure 25 shows the interconnection structure of an 8 node hypercube. In a hypercube architecture, each of the P processors has $\log_2 P$ nearest neighbor processors with which it shares a direct communications link. In order for a processor to communicate with a processor that it is not directly connected to, the communications must be routed via one or more intermediate processors. In theory, the longer the communications path and the greater the number of intermediate routing processors, the higher the communications costs. As a result, it is desirable to assign the LPs to physical processors in such a way that the number of inter-LP connections which correspond with single-hop (i.e., direct) physical connections is maximized.

To address these topological concerns, two additional features have been added to the partitioning algorithm as menu selectable options. The first deals with the order in which the LPs are built during the SDF and SBF initial partitions. This option is based upon the assumption that LP_0 will be assigned to processor 0, LP_1 will be assigned to processor 1, . . . , and LP_{P-1} will be assigned to processor P-1. When performing the initial SDF or SBF

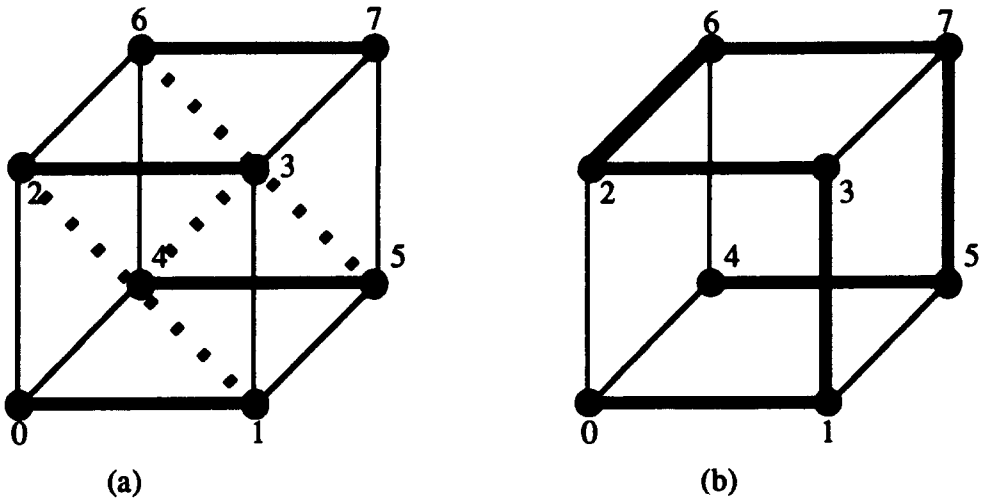


Figure 25. Topological Layout on Hypercube Connectivity Graph

partitionings, the default ordering for assigning behaviors to LPs is to begin with LP_0 and proceed in a straight numerical ordering to LP_{P-1} (e.g. 0-1-2-3-4-5-6-7).

The potential problem with this approach lies in the fact that with the SDF and SBF partitionings, there is a tendency for LP_n to have a large number of connections[12] to LP_{n+1}. However, in a hypercube architecture, direct connections are determined by the binary representation of the processor numbers, and processors n and n+1 may not be directly connected. Specifically, only those processors whose binary representations differ in a single bit position are directly connected. If processor n and processor n+1 are not directly connected, then this situation will result in an increase in the amount of multi-hop communications. For example, if LP1 has a connection to LP2 (assigned to processors 1 and 2 respectively), each message from LP1 to LP2 will traverse two physical communication links because processors 1 and 2 are not directly connected in a hypercube architecture. This is show graphically in Figure 25.a.

In an attempt to minimize the amount of multi-hop communications, an alternative LP assignment ordering is used based upon a path through the hypercube in which each subsequent processor has a binary representation that differs from the previous one by a

[12] Here, the term *connection* corresponds to an inter-behavior arc that crosses LP boundaries.

single bit position (e.g., 0-1-3-2-6-7-5-4). This is shown graphically in Figure 25.b. However, such a path only exists if the number of processors is a power of 2. In those circumstances where the number of processors is not a power of 2, the "extra" processors will be assigned in a straight numerical order. For example, the assignment ordering for 12 processors will be 0-1-3-2-6-7-5-4-8-9-10-11. Note that in a random partition, this option has no meaning and is ignored.

The second feature which has been added as a option to address topological concerns is the ability to weight inter-LP arcs based upon the number of hops in the corresponding physical communications link. When activated, this option will have no effect upon the initial SDF or SBF partitions, but will influence behavior reassignment decisions in the border-annealing process. The net effect will be a tendency to reduce the number of multi-hop communications at the potential cost of an increase in the amount of single-hop communications.

3.5 Summary

This chapter discussed the partitioning objectives of achieving a balanced computation load while minimizing the inter-process communications overhead. Equations for modeling the quality of a partition as it relates to these objectives was proposed. The model proposed for measuring the inter-process communications overhead takes into account the additional message traffic caused by the conservative null-message protocol. The communications overhead cost is combined with a factor that accounts for load imbalance to create an objective cost function for the partition. A multi-phased partitioning algorithm is then proposed with the objective of minimizing the objective cost function.

The next chapter presents the detailed implementation of the proposed partitioning algorithm, and discusses the primary test cases used in this thesis.

IV. Implementation

4.1 Overview

This chapter presents a high-level discussion of the implementation of the VHDL Graph-Partitioning Tool (GP-Tool). It also discusses the primary VHDL circuits used as test cases in order to validate the partitioning strategies implemented as part of this thesis. A complete GP-Tool user's guide and a more detailed discussion of the GP-Tool implementation can be found in Appendix C.

4.2 VSIM Graph-Partitioning Tool (GP-Tool)

The VHDL Graph-Partitioning Tool (GP-Tool) implemented in this thesis is an extension of the partitioning utility *VHDL Graph Searching Program* implemented during previous AFIT research. It was written[13] in order to provide a random distribution of the circuit behaviors among a desired number of LPs. The following sections present an overview of the additional functionality added to GP-Tool as part of this thesis. More information can be found in Appendix C.

4.2.1 Implementation Environment. GP-Tool is implemented in the Ada programming language using the Sun Ada Compiler, version 1.1. The algorithms implemented in GP-Tool are all sequential, with Sun workstations as the target platform. The original source code was written in a procedural fashion (i.e. not object-oriented), and made heavy use of Ada generics to provide the underlying data structures and data structure manipulation routines. The current version of GP-Tool is also written in a procedural fashion. In addition to allowing maximum reusability of the original source

[13] By AFIT instructor Maj Eric R. Christensen, USA.

```
9 ET_DFF_TEST_BENCH(STRUCTURAL) 0 1 2
8 ET_DFF_TEST_BENCH(STRUCTURAL) 0 3
7 ET_DFF(STRUCTURAL) 0
6 ET_DFF(STRUCTURAL) 0
5 NAND_GATE(SIMPLE) 3000000 4 7
4 NAND_GATE(SIMPLE) 3000000 5 6
3 NAND_GATE(SIMPLE) 3000000 0 2
2 THREE_INPUT_NAND_GATE(SIMPLE) 3000000 3 5
1 NAND_GATE(SIMPLE) 3000000 0 2 4
0 NAND_GATE(SIMPLE) 3000000 1
```

Figure 26. GP-Tool Input File for Edge-Triggered D Flip-Flop

code, procedural programming is the preferred methodology for routines that are computationally intensive, such as the AB border-annealing algorithm (24:24). Appendix C details the relationship between the original (unmodified) Ada packages, the modified Ada packages, and the new Ada packages which comprise the current version of GP-Tool.

4.2.2 Input and Output Files. At application startup, GP-Tool will prompt the user for the name of the desired input file. The input file can be created manually for trivially small circuits. For larger circuits, the VSIM utility `vmap` can be used to create the correct file (see Appendix B more information on `vmap`). This file lists the behaviors in the circuit and describes the behavior inter-dependency relationships. Each line in the input file must be of the following format:

`behav_id behav_name behav_delay [optional list of dependencies]`

where all values are nonnegative integers except `behav_name` which is a string of no more than 80 characters. Figure 26 shows an example input file for the edge-triggered D flip-flop of Figure 18. Note that behaviors 6 through 9 have zero delays.

There are numerous output files that can be produced by GP-Tool. Of these files, the two most important are the behavior-to-LP mapping file (`lpx.map`) and the logical

process (LP) dependency file (`lpx.arcs`) which are required for the parallel execution of VSIM. The first file maps each behavior to an "owning" LP, while the latter file defines the inter-dependency relationships of the LPs in the system.

In the original version of GP-Tool, the `lpx.map` describing the random partition was produced directly by GP-Tool, but the `lpx.arcs` file was not. Instead, an intermediate file was produced which, along with the `lpx.map` file, was used as the input to a separate utility application (`build_arc`) which produced the appropriate `lpx.arcs` file. However, the `build_arc` utility was unable to handle the large circuit description files which comprised the primary test cases used in this thesis. To circumvent this problem, the current version of GP-Tool directly produces the `lpx.arcs` file corresponding to each `lpx.map` file. The specific format specifications for the `lpx.map` and `lpx.arcs` files are discussed in section B.4.1 of Appendix B.

The other output file which is of primary interest is the partition statistics file which provides a large amount of information about the quality of the resulting partition. Among the information reported is the number of inter-component arcs, the communications cost factor (H_c), the communications distribution factor (H_d), the number of LP output lines (O_{arcs}), the load delta factor (H_b), the communications weight matrix, and a list of the behaviors assigned to each LP. This information is provided to facilitate the comparison of the quality of different partitions. This file is produced automatically for each partition generated, but is not required by VSIM.

An example partition statistics file is shown in Figure 27. This figure shows a Simple Depth-First (SDF) partition for the Wallace-Tree multiplier with 4 LPs. The top portion of the file gives the general information about the input problem-graph. Specifically, it lists the name of the input file and the number of vertices and arcs in the problem graph. The middle section lists detailed information about the partition, beginning with the name of the partitioning algorithm used. The other values presented are as follows:

- **Number of components** - Gives the number of LPs in the partition.
- **Inter-component arcs** - Gives the total number of inter-behavior arcs which cross LP boundaries.
- **Wght_Inter_LP_Arcs** - Represents the inter-component arcs with each arc multiplied by the hop-weight of the corresponding physical communications link. This is equivalent to the sum of the entries in the communications weight matrix. If all arcs are evenly weighted with a value of 1.0, this figure will be the same as the previous item.
- **Avg_Wght_Arcs** - Represents the average output arc weight for each LP (`Wght_Inter_LP_Arcs` divided by the number of LPs). This is equivalent to the average of the row sums in the communications weight matrix.
- **Stddev_Wght_Out_Arcs** - Represents the standard deviation of the output arc weights for each LP. This is equivalent to the standard deviation of the row sums of the communications weight matrix.
- **Maxdev_Wght_Out_Arcs** - Represents the maximum positive deviation of the output arc weights for each LP. This is equivalent to the maximum positive deviation of the row sums of the communications weight matrix.
- **Stddev_Wght_In_Arcs** - Represents the standard deviation of the input arc weights for each LP. This is equivalent to the standard deviation of the column sums of the communications weight matrix.
- **Maxdev_Wght_In_Arcs** - Represents the maximum positive deviation of the input arc weights for each LP. This is equivalent to the maximum positive deviation of the column sums of the communications weight matrix.

```
GRAPH INFORMATION            - wallace.vmap
-------------------------------------------------------------------
The number of vertices in this graph is    : 1050
The number of arcs in this graph is        : 1770

PARTITION INFORMATION        - Simple Depth-First (SDF) Partitioning
-------------------------------------------------------------------
Number of components : 4
Inter-component arcs : 312

Wght_Inter_LP_Arcs   :  312.0
Avg_Wght_Arcs        :   78.0
Stddev_Wght_Out_Arcs :   66.5
Maxdev_Wght_Out_Arcs :   89.0
Stddev_Wght_In_Arcs  :   46.6
Maxdev_Wght_In_Arcs  :   49.0
Comm_Cost_Factor     :  17.63 %
Comm_Dist_Factor     : 114.10 %
LP_Output_Lines      : 11
Lookahead_Factor     : 0.667

Avg_Comp_Load        :  262.5
Stddev_Comp_Load     :    0.6
Maxdev_Comp_Load     :    0.5
Load_Delta_Factor    :   0.19 %

The LP loads are :
  263   263   262   262

The communications weight matrix is :
   0.0   2.0   0.0   5.0         7.0
  56.0   0.0   3.0   2.0        61.0
  21.0  48.0   0.0   8.0        77.0
  50.0  40.0  77.0   0.0       167.0


 127.0  90.0  80.0  15.0   =>  312.0

The total partition cost is                 :   3.65

The predicted speedup is                    :   3.01
Speedup prediction parameters :
    alpha : 100.0000000000
    beta  :  1.0000000000
    gamma :  0.0900000000
```

Figure 27. Wallace-Tree SDF Partition Statistics File for 4 LPs

- **Comm_Cost_Factor** - This factor is the H_c discussed in section 3.3.2.1. It represents `Wght_Inter_LP_Arcs` divided by the total number of inter-behavior arcs in the input problem-graph.
- **Comm_Dist_Factor** - This factor is the H_d discussed in section 3.3.2.2. It represents the difference between `Maxdev_Wght_Out_Arcs` and `Avg_Wght_Arcs` divided by `Avg_Wght_Arcs`.
- **LP_Output_Lines** - This factor is the O_{arcs} discussed in section 3.3.2.4, and represents the number of arcs in the LP connectivity graph. It is equivalent to the number of non-zero entries in the communications weight matrix and the number of output lines specified in the `lpx.arcs` file.
- **Lookahead_Factor** - This factor is the L_{arcs} discussed in section 3.3.2.4, and provides a measure of the amount of lookahead in the `lpx.arcs` file (the smaller the value, the larger the average lookahead).
- **Avg_Comp_Load** - This factor is the L_{avg} discussed in section 3.3.1, and represents the average computation load of all the LPs. Since all behaviors are equally weighted, this is equivalent to the number of vertices in the problem-graph divided by the number of LPs.
- **Stddev_Comp_Load** - Represents the standard deviation of the LP computation loads.
- **Maxdev_Comp_Load** - Represents the maximum positive deviation of the LP computation loads.
- **Load_Delta_Factor** - This factor is the H_b discussed in section 3.3.1. It represents `Maxdev_Comp_Load` divided by `Avg_Comp_Load`.
- **LP Loads** - Lists the computation load (i.e. number of behaviors) assigned to each LP beginning with LP0 and proceeding in numeric order from left to right.

- **Communications Weight Matrix** - This is the n x n communications weight matrix shown in Figure 20 with an additional column to hold the row sums, and an additional row to hold the column sums. The bottom right entry holds the sum of all n x n entries and corresponds to the value `Wght_Inter_LP_Arcs`.
- **Total Partition Cost** - This is the value of the objective cost function of section 3.3.4.1.
- **Predicted Speedup** - This is the value of the estimated speedup equation of section 3.3.4.2.
- **Speedup Prediction Parameters** - Alpha and beta represent the coefficients used to balance the communications and load imbalance factors of the objective cost function. Gamma is used as the coefficient to the total partition cost in the speedup estimate function.

The bottom portion of the partition statistics file lists the behaviors assigned to each LP and the number of arcs that are local[14] to that LP, and is omitted from Figure 27.

4.2.3 Data Structures. The implementation of the partitioning algorithms is heavily dependent upon the data structures used to represent the problem-graph for the circuit being simulated. Several modifications to the underlying data structures used in the original version of GP-Tool have been implemented in order to improve algorithm efficiency and to facilitate the implementation of more sophisticated partitioning algorithms.

In the original GP-Tool, a graph was represented by a set of vertex records and a set of arc records. In addition, each vertex record contained its own set of arc records for arcs that originated from that vertex. Each behavior in the circuit being simulated corresponds

[14] A *local arc* is defined as one that is between two behaviors assigned to the same LP.

```
-- Declaration of the Vertex Nodes & Instantiation of Generic Set pkg

  type Vertex_Node is
    record
      The_Item        : Item;
      The_Arcs        : Arc_Set.Set;      -- set of output arcs
      Reference_Count : Natural := 0;     -- number of input arcs
      Next            : Vertex;           -- for garbage collection
    end record;
  type Vertex is access Vertex_Node;
  package Vertex_Set is new Set_Iterator(Item => Vertex);


-- Declaration of the Arc Nodes & Instantiation of Generic Set pkg

  type Arc_Node is
    record
      The_Attribute   : Attribute;
      The_Source      : Vertex;     -- pointer to source vertex
      The_Destination : Vertex;     -- pointer to destination vertex
      Next            : Arc;
    end record;
  type Arc is access Arc_Node;
  package Arc_Set is new Set_Iterator(Item => Arc);


-- Declaration of the Graph Type

  type Graph is
    record
      The_Vertices : Vertex_Set.Set;
      The_Arcs     : Arc_Set.Set;
    end record;
```

Figure 28. Original GP-Tool Graph Data Structures

to a unique vertex record, and each dependency between behaviors corresponds to a unique arc record.

The original data structure declarations are shown in Figure 28. With these data structures, procedures were provided to take a graph as input and iterate through the set of vertices or the set of arcs which comprised the graph. A procedure was also provided to take a single vertex as input and iterate through its set of output arcs. As shown in Figure 28, each arc maintained pointers to its source vertex and its destination vertex. Thus, it

```
-- Declaration of the Vertex Nodes & Instantiation of Generic Set pkg

  type Vertex_Node is
    record
      The_Item        : Item;
      The_Arcs        : Arc_Set.Set;  -- set of output arcs
      Incoming_Arcs   : Arc_Set.Set;  -- set of input arcs
      Reference_Count : Natural := 0; -- number of input arcs
      Parent          : Vertex; -- ptr to prev member of group
      Child           : Vertex; -- ptr to next member of group
      Next            : Vertex; -- for garbage collection
    end record;
  type Vertex is access Vertex_Node;
  package Vertex_Set is new Set_Iterator(Item => Vertex);


-- Declaration of the Arc Nodes & Instantiation of Generic Set pkg

  type Arc_Node is
    record
      The_Attribute   : Attribute;
      The_Source      : Vertex;    -- pointer to source vertex
      The_Destination : Vertex;    -- pointer to destination vertex
      Next            : Arc;
    end record;
  type Arc is access Arc_Node;
  package Arc_Set is new Set_Iterator(Item => Arc);


-- Declaration of the Graph Type

  type Graph is
    record
      The_Vertices : Vertex_Set.Set;
      The_Arcs     : Arc_Set.Set;
    end record;
```

Figure 29. Modified GP-Tool Graph Data Structures

was possible to traverse the graph in the forward direction (i.e. following arcs from tail to head), but not in the reverse direction. This is because the set of input arcs was not maintained by each vertex record.

To alleviate this problem, the original version of GP-Tool built two separate graphs: an "adjacency graph" which had the arcs in their forward directions, and a "dependency graph" which was identical except that the direction of the arcs was reversed. This dual

```
type Process_Node_Type is
  record
    Process_Id      : Natural;
    Label_Name      : String80.String_Type;
    The_Delay       : Natural := 0;
    The_LP          : Natural := 0;
    Group_Size      : Natural := 1;
    Group_Num_Arcs  : Natural := 0;
  end record;
```

Figure 30. `Process_Node_Type` Data Structure

graph approach was adequate for many applications (such as finding the strong components), but it had several disadvantages. First, it doubled the amount of memory required to maintain the graph information. Second, it required twice as long to build the graph from the input file. Third, and most significantly, there was no way to simultaneously iterate both the input and output arcs of a given vertex without iterating through the entire vertex set of the dual graph to locate the matching vertex record. The ability to perform this last function is critical to being able to efficiently prioritize behaviors for potential LP reassignment during the border annealing process.

To provide the needed functionality, the vertex record was modified to include a set of incoming arcs in addition to the set of outgoing arcs. The appropriate procedure to iterate this new set of incoming arcs was also added. Together, these changes obviated the need to maintain two separate graphs in memory. The modified data structure declarations are shown in Figure 29.

In addition to adding a set of incoming arcs to the vertex record, two additional fields (`Parent` and `Child`) were added to allow groups of vertices to be linked together in a doubly-linked list fashion. These fields are used to link together the behaviors that have been assigned to the same LP. In doing so, it is possible to quickly iterate through all of the behaviors that have been assigned to a given LP to gather pertinent information, such as the number of arcs that are local to that LP, without the need to maintain complex

external data structures. To facilitate the recording of partition information in the graph data structure, each vertex keeps track of which LP it is currently assigned to. In addition, one vertex in each LP is arbitrarily chosen to be the "list head" for that LP, so called because its `Parent` field points to a null vertex placing it at the head of the doubly-linked list which links together the members of the LP. For easy access, the size of the LP and the number of local arcs for that LP are recorded in the list head (referred to as the "head vertex" for that LP). To track all of this information, the record data structure shown in Figure 30, referred to as the `Process_Node_Type`, is used as the "Item" type in the Vertex_Node record of Figure 29.

In addition to providing a place to record LP information, the `Process_Node_Type` data structure is where the behavior specific information is recorded. Specifically, the process id number, the behavior's label name, and the behavior delay are recorded. It should be noted that to minimize data duplication, only the head vertex for each LP maintains the information regarding the LP's size and number of local arcs. These fields are simply ignored if the vertex is not the head vertex.

4.2.4 Menu Structure. The current version of GP-Tool uses a two-level menu structure. The GP-Tool main menu is shown in Figure 31. Items 1 through 4 allow the user to produce various output files not directly related to the partitioning files, and are discussed further in Appendix C.

Item 5 on the main menu takes the user to the behavior mapping sub-menu shown in Figure 32, from which the user can select the desired partitioning algorithm as well as modify various user defined partitioning parameters. Items 4-5 allow the user to select an AB Annealing partition using the depth-first, breadth-first, or random partitioning algorithms respectively to provide the initial partition. Reference Appendix C for more detailed information on the available menu options.

```
********************** GP-TOOL MAIN MENU ************************

Select one of the following operations:
1 : Generate Delay and Adjacency Information File
2 : Generate SGE Data File
3 : Generate Topological Sort File
4 : Generate Strong Components File
5 : Generate Behavior to Logical Process (LP) Mapping File(s)
0 : Quit GP-Tool


Enter your menu choice now:
```

Figure 31. GP-Tool Main Menu

4.2.5 Strong Component Search. In directed graph terminology, a strongly connected component is defined as a maximal set of vertices with the property that there is a path between any two vertices in the set (10:488). In terms of a problem graph for a VHDL circuit simulation, a strongly connected component represents a complete feedback loop, such as the one in Figure 33. Such a feedback loop creates a circular dependency during simulation. It is common for such feedback loops in digital circuits to

```
************* GP-TOOL BEHAVIOR MAPPING MENU *************

Select one of the following operations:
1 : Generate Random Partitioning File
2 : Generate Simple Depth-First Partitioning File
3 : Generate Simple Breadth-First Partitioning File
4 : Generate AB1-Annealing Partitioning File
5 : Generate AB2-Annealing Partitioning File
6 : Generate AB3-Annealing Partitioning File
7 : Turn the .MAP and .ARCS output OFF
8 : Modify the Cost Function Parameters
9 : Return to Main Menu
0 : Quit GP-Tool


Enter your menu choice now:
```

Figure 32. GP-Tool Mapping Sub-Menu

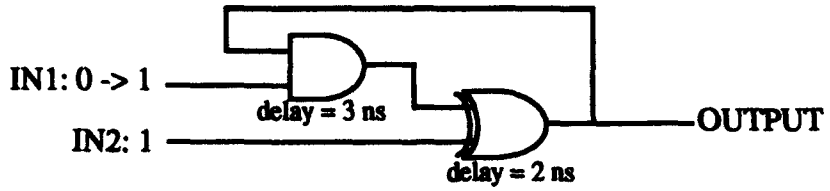


Figure 33. Example Feedback Loop - Simple Oscillator

involve a large number of signal state changes. By isolating the feedback loop on a single LP, it may be possible to reduce the amount of inter-LP message traffic.

For example, consider the simple oscillator circuit in Figure 33. Signal IN2 is tied high, while signal IN1 is initially low and is taken high to begin the output oscillation. When IN1 is taken high, the output signal will change states every 5 ns (the sum of the delays of the AND and XOR gates). If the AND and XOR gates are assigned to different LPs, two inter-LP messages (AND to XOR, and XOR to AND) will result for each change of the output state. If the AND gate is placed on the same LP as the XOR gate, however, these messages will no longer be necessary (all communications will take place via the local behavior and active signal lists).

The strong components of the problem graph are found using the following algorithm (10:489):

- Perform a depth-first search on the input graph with the arcs in the reverse direction, keeping track of the order in which the vertices are finished. A vertex is finished when all paths leaving the vertex have been fully explored.
- Perform a second depth-first search on the input graph with the arcs in the forward direction. However, begin new depth-first trees by considering the vertices in the reverse order of their finishing times in the initial search of the previous step. Keep track of the depth-first trees of this second search.
- Output the depth-first trees from the second search. Each one of these trees corresponds to a strongly connected component of the input graph.

4.2.6 Simple Depth-First (SDF) Partition. The implementation of the Simple Depth-First (SDF) partitioning algorithm is based upon the depth-first search routine used in finding the strong components. The algorithm consists of the following steps:

- Calculate the expected size of the current LP by dividing the number of unassigned vertices by the number of LPs remaining to be filled, rounding up to the nearest integer. Reset the vertex counter to zero.
- While the vertex counter is less than the expected size of the current LP, traverse the graph in a depth-first manner with the arcs in the forward direction using a source vertex[15] as the starting point. As previously undiscovered vertices are visited, assign them to the current LP, mark them as discovered, and increment the vertex counter. If a newly discovered vertex is part of a strong component, assign the entire strong component to the current LP and increment the vertex counter by the size of the strong component. Note that this may put the vertex counter over the limit set by the size of the LP calculated in the previous step. Finding the strong components of the graph prior to performing the SDF partition is optional. In the current version of GP-Tool, the SDF partition by itself does not consider strong components. However, when the SDF partition is used as the initial partition for the AB-Annealing algorithm, a strong component search is performed as the first step in the partitioning process.
- If the current depth-first search tree is completed before the current LP has reached its target size, begin a new search by choosing another undiscovered source vertex as the next starting point. If no more source vertices remain, choose an arbitrary undiscovered vertex as the next starting point.

[15] A source vertex is one with no inputs.

- If the current LP reaches its target size before the current depth-first search tree is completed, the search is terminated and the process repeats starting again at step one for the next LP. A new depth-first search tree is started for each successive LP in an attempt to increase the probability of assigning a complete depth-first search tree to the LP. This is desirable because each depth-first tree represents a set of dependent tasks, and assigning dependent tasks to the same LP will reduce the inter-LP communications overhead.

This algorithm is similar to the Depth-First Breadth-Next (DFBN) algorithm discussed in section 2.4.8, except that load balancing is considered in the SDF algorithm whereas it is not addressed in the DFBN algorithm. Some characteristic traits of the partitions generated by the SDF algorithm are as follows:

- The first LP will contain long paths of dependent behaviors with a large number of local arcs.
- Each successive LP will tend to have shorter paths of dependent behaviors than the preceding LP as it gets more difficult to find long paths of dependent behaviors which are not yet assigned to an LP.
- The final LP will consist of the fragments of the problem graph that were not assigned to a previous LP, and will tend to have a relatively small number of local arcs.

4.2.7 Simple Breadth-First (SBF) Partition. The implementation of the Simple-Breadth First (SBF) partitioning algorithm is identical to that of the SDF algorithm with the following exceptions:

- The problem graph is traversed in a breadth-first manner.

- When an LP is full, the graph traversal for the subsequent LP assignment picks up where the previous one had left off. The breadth-first search tree is not terminated prematurely.

4.2.8 AB Border Annealing Algorithm. The implementation of the AB Border Annealing algorithm corresponds to the steps discussed in section 3.4.3. However, before beginning the first iteration, the graph is evaluated to initialize several data structures with statistical information concerning the state of the initial partition. The most significant of these data structures is the initial value of the communications cost sub-function:

$$H_n H_c (1 + H_d)$$

where

$$H_n = L_{arcs} O_{arcs} = (1.0) O_{arcs} = O_{arcs}$$

because the value of L_{arcs} is not known until the final state of the partition has been reached. It is calculated as part of the routine that prints the corresponding `lpx.arcs` file. The current algorithm for computing L_{arcs} is time consuming and including it in each step of the annealing process would render the algorithm computationally infeasible. The algorithm for computing L_{arcs} is discussed further in section 5.4.

In addition to ignoring the effect of the lookahead, an additional option has been added to the annealing input parameters:

- `Ignore_Comm_Dist_Factor` - boolean value that allows the factor H_d to be ignored when computing the value of the communications sub-function during the annealing process.

When `Ignore_Comm_Dist_Factor` is true, the communications cost is calculated as:

$$H_n H_c$$

During data analysis, it was discovered that in some circumstances, the annealing algorithm has a tendency to accept a small increase in O_{arcs} in order to gain a decrease in H_d. However, the resulting partition resulted in a decrease in simulation performance over the initial partition, indicating that the increase in O_{arcs} dominated the decrease in H_d. The option `Ignore_Comm_Dist_Factor` was included to force the algorithm to accept an increase in H_d in order to decrease the remaining portion of the cost function (i.e., $H_n H_c$).

Once the initial communications costs have been calculated, the annealing process begins as shown in Figure 34. The annealing queue is filled by the procedure `Prioritize_And_Queue` using the criteria discussed in section 3.4.3.1. Once the queue has been filled, vertices are removed in priority order and considered for LP reassignment by the procedure `Consider_Vertex`.

The procedure `Consider_Vertex` initializes several array data structures with information concerning the impact on the objective cost function of reassigning the given vertex to each viable destination LP. Only those LPs which contain behaviors that are directly connected to the given vertex are considered viable. The specific data structures maintained are:

- **`Input_Arcs_Array`** - Records the number of input arcs to the subject behavior that originate from behaviors in each of the other LPs.
- **`Output_Arcs_Array`** - Records the number of output arcs from the subject behavior that go to behaviors in each of the other LPs.
- **`Wght_Arcs_Array`** - Records the net change in the value of `Wght_Inter_LP_Arcs` (sum of the inter-component arcs with each arc multiplied by the corresponding hop weight). Used to calculate the change to H_c.
- **`Maxdev_Comm_Array`** - Records the net change in the value of `Maxdev_Wght_Out_Arcs`. Used to calculate the change to H_d.

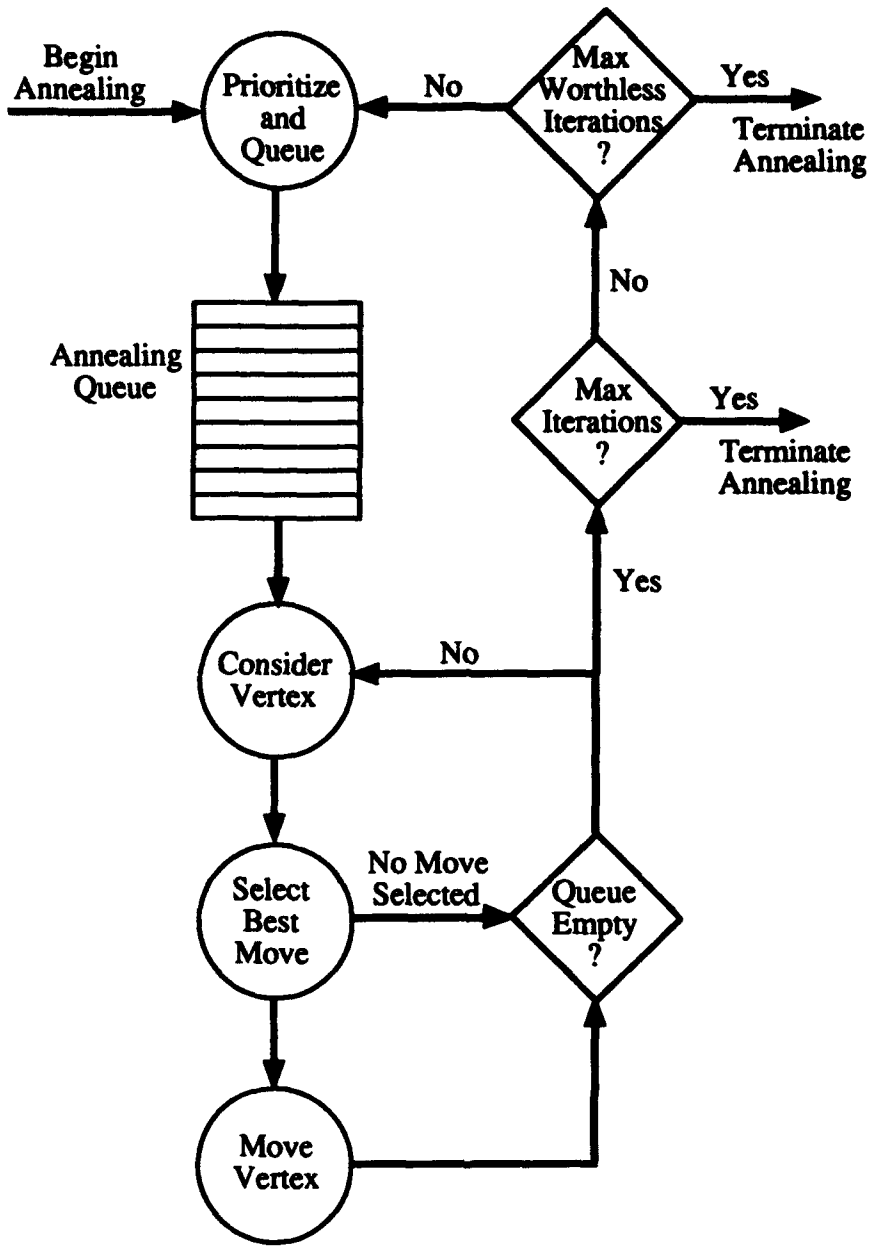


Figure 34. AB Border Annealing Algorithm Cycle

- **Stddev_Comm_Array** - Records the net change in the value of `Stddev_Wght_Out_Arcs`. Used as a tie breaker, if necessary.
- **Maxdev_Load_Array** - Records the net change in the value of `Maxdev_Comp_Load`. Used to calculate the change to H_b.

- **Output_Line_Array** - Records the new value of `LP_Output_Lines` (number of arcs in the LP connectivity graph). Used to calculate the change to H_n.

Each of these data structures is a one-dimensional array indexed by the destination LP number. If the destination LP is not a viable destination, the corresponding values in the `Wght_Arcs_Array`, `Maxdev_Comm_Array`, and `Stddev_Comm_Array` are set to an arbitrarily large value (e.g. 2 times the number of arcs) to effectively eliminate these LPs from move consideration.

The first two data structures, `Input_Arcs_Array` and `Output_Arcs_Array`, are initialized once at the beginning of the procedure `Consider_Vertex`. In a worst case scenario, a given behavior has input arcs from all other LPs and output arcs to all other LPs. In this situation, it would take O(N) operations to initialize these data structures (where N is the number of behaviors in the graph). However, on average, each behavior will have only E/N input arcs and E/N output arcs (where E is the number of arcs in the graph). For a given circuit, E and N are fixed and have a relatively small ratio. Thus, on average, it takes only $O(2E/N) = O(1)$ operations to initialize these data structures.

For viable destination LPs, however, the net change to the communications cost factors must be calculated. Although the communications costs between LPs are recorded in an P^2 data structure (the `Comm_Weight_Matrix`), where P is the number of LPs, only those rows and columns associated with the source and destination LP will be affected by the move. Thus, the net change to the communications cost factors for a particular destination LP are calculated in O(P) time. Since there are (P-1) potential viable destination LPs, the upper limit of the running time order of the `Consider_Vertex` procedure is $O(P(P-1)) = O(P^2)$ (assuming O(1) average time to calculate the input/output dependencies of a viable destination LP as discussed in the previous paragraph). However, as P is increased, it is reasonable to expect that only a small fraction of the LPs

will be viable destinations for the average vertex, making the average running time for `Consider_Vertex` approximately O(P).

The data structures initialized by the procedure `Consider_Vertex` are passed to the procedure `Select_Best_Move` which evaluates the viable moves to find the one which will result in the smallest value of the communications cost sub-function:

$$H_n H_c (1 + H_d)$$

or

$$H_n H_c$$

if `Ignore_Comm_Dist_Factor` is set to true. If the destination LP with the smallest communications cost sub-function value will result in a change to H_b that will put it over the maximum value (`Load_Imbal_Tol`), the selected move is discarded and the next best move is sought. In no case will an increase in the communications cost sub-function be allowed. Thus, it is possible that no move will be selected. The procedure's running time is O(P) since each LP must be considered.

If a move is selected, it is carried out by the procedure `Move_Vertex`. The move involves an update to the partition statistics values to record the new cost factors, as well as a series of simple list inserts and deletes to assign the vertex to the new LP. The procedure `Move_Vertex` has a running time of O(P) since the communications weight matrix must be updated to contain the new values for the rows and columns associated with the source and destination vertex.

If the annealing queue is not empty, the next vertex is removed and the consideration process is repeated. If the annealing queue is empty, the current iteration is completed. If the maximum number of iterations has not been reached, the value of the communications cost sub-function is compared to the value computed at the end of the previous iteration (or at the beginning of the algorithm for the first iteration). If there was no net improvement, the iteration is considered "worthless." If there have been

`Max_Worthless_Iter` consecutive iterations that were worthless, the annealing process is terminated.

With the streamlined implementations of the procedures `Consider_Vertex` and `Move_Vertex`, the most time consuming portion of the annealing cycle appears to be the procedure `Prioritize_And_Queue`. This appears to be due to the linear data structures used to provide priority queue management. A splay tree queue implementation may provide a higher level of efficiency.

4.3 Test Cases

4.3.1 Wallace-Tree Multiplier. Prior to this thesis effort, the wallace tree multiplier was the largest VHDL circuit simulated in parallel at AFIT with the VSIM simulator. The multiplier takes two eight bit vectors as input and produces a single twelve bit product vector as output (4:131). The resulting problem graph consists of 1,050 behaviors and 1,770 inter-behavior arcs. The simulation runs from 0 to 2000 ns.

4.3.2 Associative Memory Array. The associative memory array circuit consists of a 16 x 16 memory array, associated control registers, and 68 vector testbench. The associative memory is currently the largest circuit simulated with VSIM. The resulting problem graph consists of 4,243 behaviors and 9,312 inter-behavior arcs. The testbench consists of the following actions:

1. Write to all memory words in order from word 0 to word 15 (16 writes).
2. Read all memory words in order from word 0 to word 15 (16 reads).
3. Search for certain pre-specified patterns (16 searches).
4. Read from all memory words in an arbitrary order (16 reads).
5. Perform read operations with multiple words selected (4 reads).

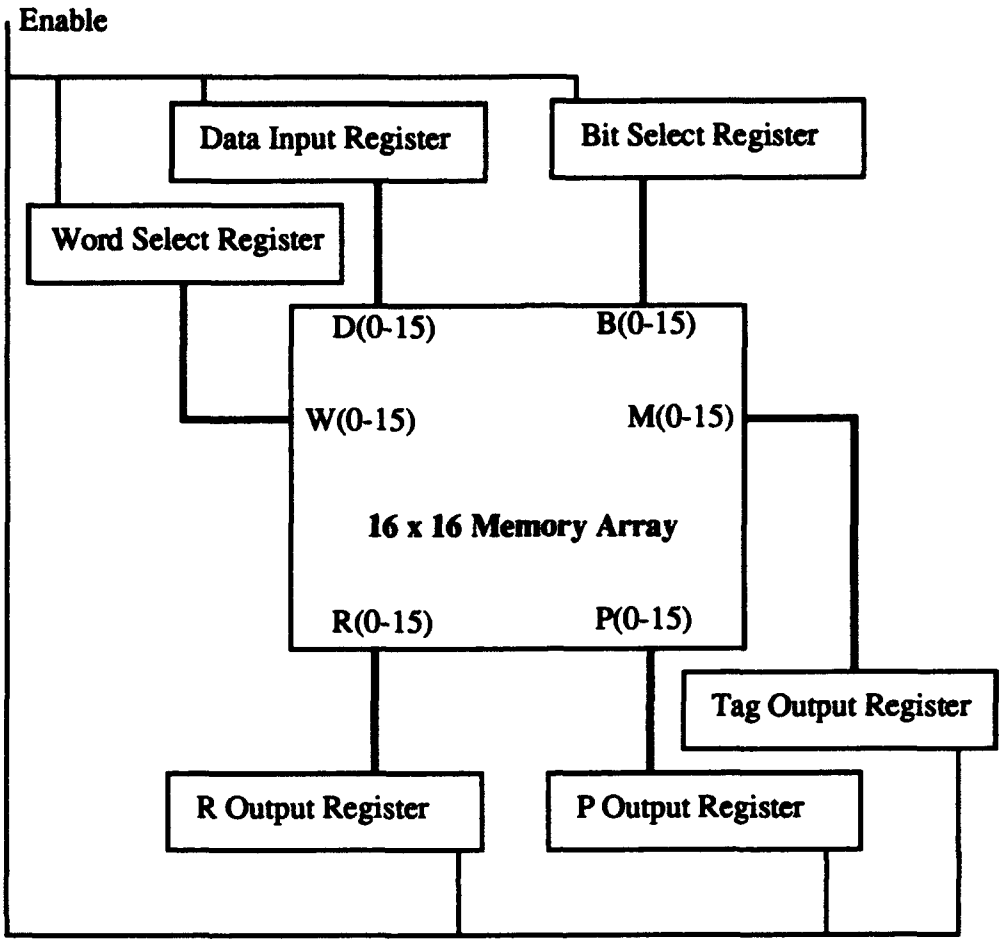


Figure 35. Associative Memory Array

The circuit is built in a hierarchical manner to allow for easy transformation and compilation with VSIM. When written to run with the Synopsis commercial VHDL simulator, the simulation ran from 0 to 8000 ns. However, VSIM has a limit of approximately 2000 ns because of the data type used to represent the simulation clock. To get around this problem, all time units in the associative memory VHDL source code were changed to picoseconds. Thus, the simulation runs from 0 to 8000 ps (8 ns).

A block diagram for the associative memory circuit is shown in Figure 35. It should be noted that all three input registers and all three output registers are clocked by the same enable signal as a matter of design convenience. A result of this is that during

memory writes, the value of the tag output register oscillates rapidly. This has the effect of adding a large number of events to the simulation, slowing down the simulation. Observation of the simulation shows that the simulation progresses slowly until the writes are completed (at time 2000 ps), at which time it begins to progress at a much faster pace. The correctness of the output remains unaffected since the content of the tag register is not relevant during a memory write.

V. *Methodology and Results*

5.1 Overview

This chapter presents a discussion of the performance of the VSIM test cases discussed in the previous chapter. Four different partitionings created with the GP-Tool utility are used. Specifically, each circuit was simulated with a random partition, a simple depth-first (SDF) partition, and a simple breadth-first (SBF) partition. The best of these three partitions was then used as the initial partition for the AB border-annealing algorithm to create a fourth partition.

The primary performance measurements were taken on the 8-node iPSC/2. All speedup calculations use a single-LP partition as the performance baseline. Each circuit was simulated with 2, 3, 4, 5, 6, 7, and 8 LPs for each of the four partition types. For the wallace tree, the performance measurements and message counts for each configuration are calculated from the average of 20 simulation trials. For the associative memory array, only 10 simulation trials were run for each configuration due to the large amount of time required for each trial.

The results of each circuit are discussed in terms of the resulting speedup, the inter-LP message traffic, and the corresponding partition statistics. The inter-LP message traffic is analyzed in terms of message traffic originating from each LP. Tables containing the simulation trial results and partition statistics for each combination involving 1, 2, 4, or 8 LPs are included in Appendix D along with supplemental inter-LP message traffic charts.

In addition to the iPSC/2 results, the wallace tree multiplier was run on an iPSC/860 hypercube using up to 32 nodes. These results are presented briefly in section 5.6.

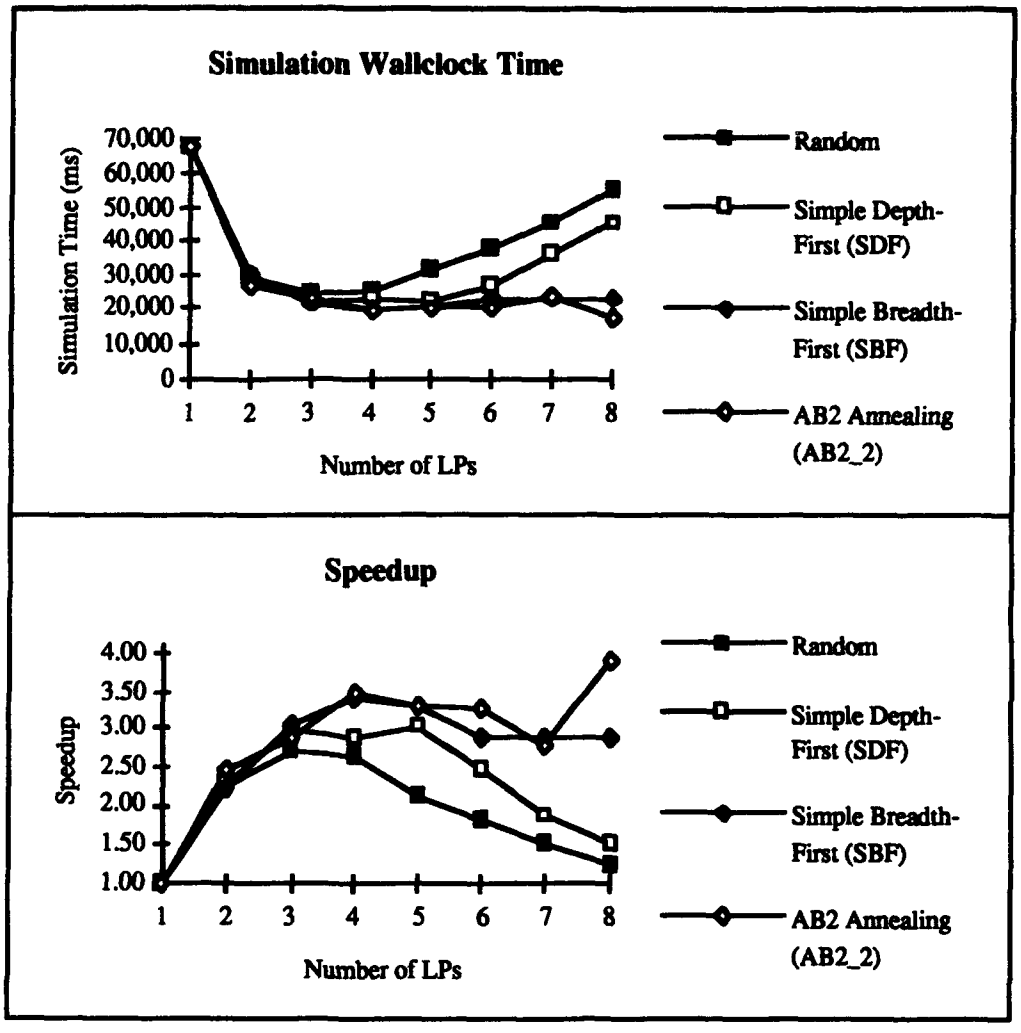


Figure 36. Wallace Tree Speedup Results Comparison

5.2 Speedup Results

5.2.1 Wallace-Tree Multiplier. The wallace tree speedup results are shown in Figure 36. All speedup calculations are in reference to the single LP case which required an average of 67,947 ms to complete. Figure 37 compares the number of LP output lines, the number of inter-LP arcs, and the communications distribution factor for each of the partitioning algorithms, while Figure 38 shows the corresponding message counts. The message counts are shown in terms of real, null, and total messages sent from all LPs.

As shown in Figure 36, the random partitioning speedup peaks at 2.72 with 3 LPs and declines to 1.23 with 8 LPs. Observation of Figures 37 and 38 shows that the reason for

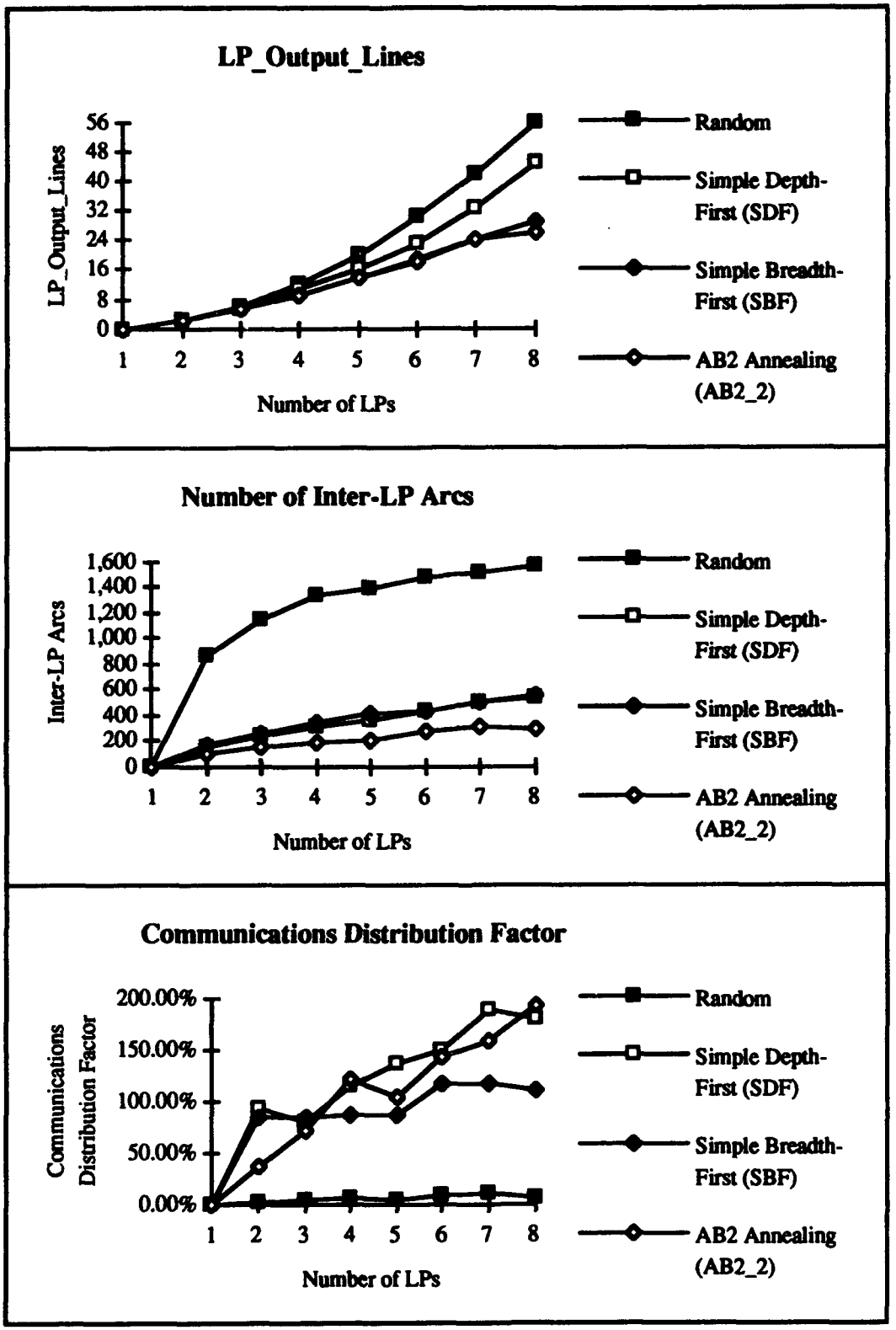


Figure 37. Wallace Tree Partition Statistics Comparison

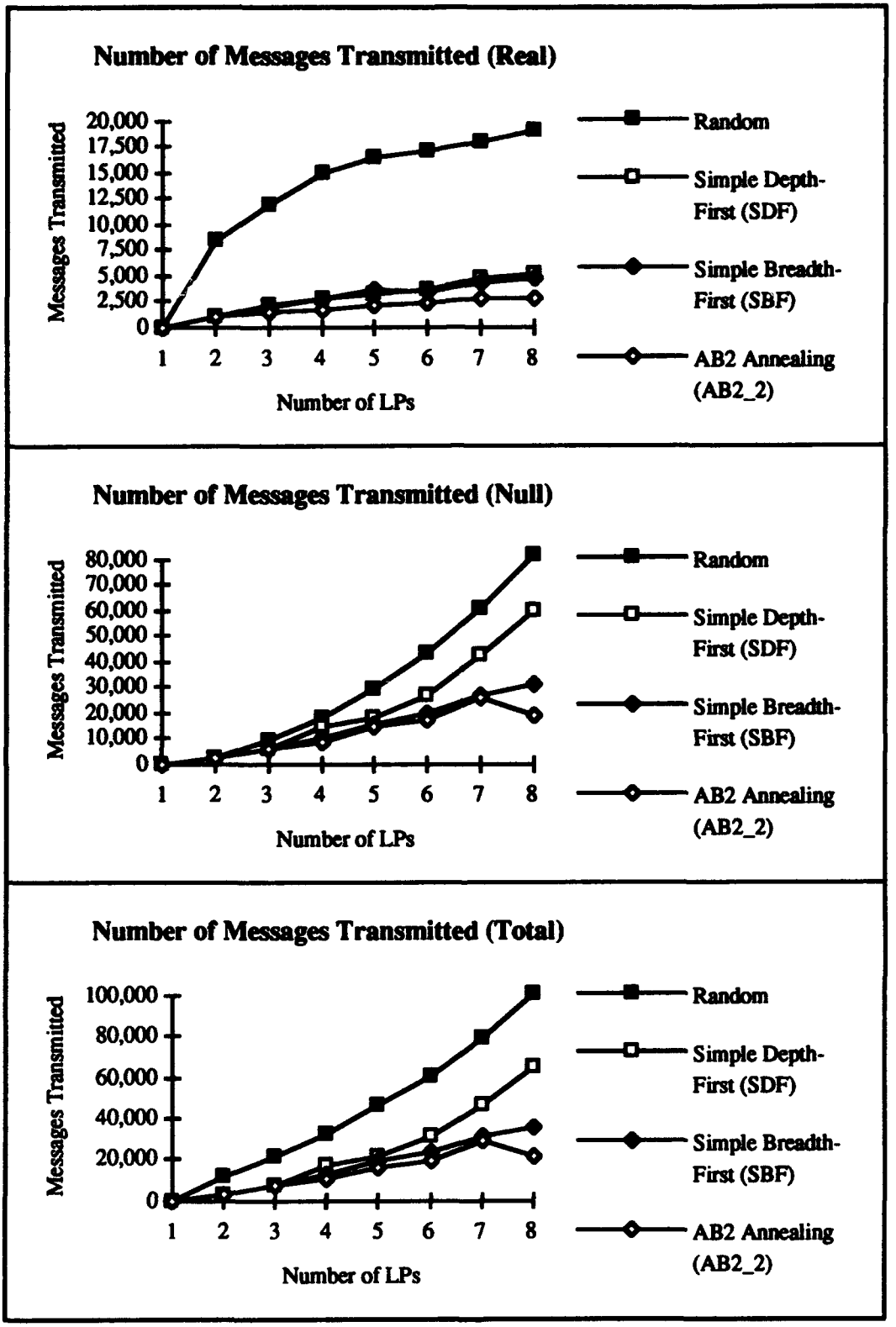


Figure 38. Wallace Tree Inter-LP Message Traffic Comparison

this sharp drop-off in speedup is due to a dramatic increase in the inter-process communications overhead as the number of LPs is increased. Specifically, Figure 37 shows that for 8 LPs, the random partition has the maximum of 56 LP output lines and a total of 1,558 of 1,770 arcs (88%) that cross the LP boundaries. Comparing these curves to Figure 38 shows the direct relationships between the number of LP output lines and the number of null messages transmitted, and between the number of inter-LP arcs and the number of real messages transmitted.

Note that the random partitioning real message curve (Figure 38) decreases in slope as the number of LPs is increased. This occurs as the number of random partition inter-LP arcs quickly approaches its theoretical maximum (e.g. 49% with 2 LPs and 75% with 4 LPs). This correlates directly to the number of real messages approaching its maximum limit at approximately the same rate. The maximum number of real messages is determined by the number of actual signal changes in the simulation. If 100% of the arcs cross LP boundaries, then every signal change in the simulation will result in the transmission of a real message.

On the other hand, the slope of the null message curve for random partitioning shows a trend of increasing as the number of LPs is increased. This follows from the direct relationship between the number of null messages and the number of LP output lines along with the fact that the random partitioning algorithm has a tendency to produce the maximum of P(P-1) LP output lines. The theoretical limit on the number of LP output lines is the same as the number of inter-LP arcs. However, the limiting factor of P(P-1) means that a larger number of LPs will be required for the actual number of LP output lines to approach its maximum. Furthermore, while the number of inter-LP arcs and LP output lines share a common theoretical maximum, the maximum number of null messages may be much higher than the maximum number of real messages. This is due to several factors. First, each real message transmitted may result in the transmission of

multiple null messages. Second, null messages are transmitted over all output lines each time an LP must block for input. Finally, the number of null messages is partially dependent upon the amount of lookahead in the corresponding `lpx.arcs` file.

An important result of this is that as the number of LPs is increased, the null message communications overhead required to avoid deadlock begins to dominate the overhead from the real message traffic. For example, the approximate null to real message ratios for random partitioning with 2, 4, and 8 LPs are 1:4, 1:1, and 4:1 respectively.

The SDF partitioning speedup curve is slightly better, peaking at 3.04 with 5 LPs and decreasing to 1.50 with 8 LPs. Looking at Figure 37, the most notable improvement between the random and SDF partitions is in the number of inter-LP arcs. With the deliberate depth-first partitioning algorithm, the number of inter-LP arcs approaches its maximum at a much slower rate, reaching only 30% with 8 LPs (vs. 88% for the random partition). Figure 38 shows how this improvement translates directly into a similar improvement in the number of real messages transmitted. For example, with 8 LPs, there is a 73% reduction in the number of real messages. In addition, the SDF partitioning algorithm reduced the number of LP output lines to only 45 with 8 LPs (vs. 56 for the random partition). Again, Figure 38 shows how this improvement translates directly into a similar improvement in the number of null messages transmitted (e.g., 27% reduction in the number of null messages with 8 LPs). It should be noted that an increase in the amount of lookahead in the `lpx.arcs` files and the decrease in the number of real messages also contributed to the decrease in the number of null messages. The increased lookahead is due to the ability of the SDF algorithm to assign relatively long chains of dependent behaviors to the same LP. In many cases, this will increase the minimum path time through the LP.

Even with this reduction in the number of null messages, however, the shape of the null message curve is still proportional to P(P-1). As a result, the null message overhead

still dominates the real message overhead as the number of LPs increases. In fact, due to the reduction in the number of real messages, fewer LPs are required until null messages begin to dominate real messages. For example, the null to real message ratio for the SDF partition begins at 2:1 with 2 LPs, and increases to 11:1 for 8 LPs. This domination of the null messages offsets the apparent benefits gained by the decrease in the real message traffic. For example, with 8 LPs, there was a 73% reduction in real messages, but only a 36% reduction in total messages. The continued high communications cost is the primary reason that the speedup curve for the SDF partition drops off rapidly with more than 5 LPs.

An additional factor which appears to limit the speedup gains of the SDF partition is the relatively high value of H_d as shown at the bottom of Figure 37. The inter-LP arcs of the random partition are relatively evenly distributed among each of the LPs. However, the SDF algorithm tends to result in partitions in which a relatively large portion of the inter-LP arcs will originate from a single LP. The relationship between H_d, the inter-LP message traffic, and the resulting speedup is discussed further in section 5.4.

The SBF partition speedup curve is even better than the SDF curve, peaking at 3.43 and declining at a much slower rate to 2.86 with 8 LPs. While Figure 37 shows that both algorithms result in partitions with approximately the same number of inter-LP arcs, the SBF algorithm is able to take advantage of the feed-forward nature of the wallace-tree circuit and produce a circuit with a significantly smaller number of LP output lines (e.g., 29 with 8 LPs vs. 45 for the SDF partition). As shown in Figure 38, this corresponds to a slower growth rate in the null message curve. Still, with the corresponding reduction in real message traffic, the communications overhead is dominated by the null message traffic with a null to real message ratio of 2:1 with 2 LPs and growing to 6:1 with 8 LPs.

Since the SBF partition provided the best speedup results, it was used as the initial partition to the AB annealing algorithm to get the fourth speedup curve shown in Figure 36. The following input parameters were used to the annealing algorithm:

```
Num_Iterations        - 500      Ignore_Comm_Dist_Factor - true
Max_Worthless_Iter    - 25       Topological            - false
Load_Imbal_Tol        - 1.5%     Hop_Weights            - all 1.0
```

With `Ignore_Comm_Dist_Factor` set to false, the algorithm reduced H_d and H_c at the expense of a slight increase in H_n. The result was a reduced and more evenly distributed real message communications load that was overwhelmed by an increased null message communications load resulting in no net improvement in the speedup curve. By setting `Ignore_ Comm_Dist_Factor` to true, the algorithm has a tendency to reduce H_n and H_c at the expense of a potential increase in H_d. This effect can be seen by comparing the SBF and AB Annealing curves in Figure 37.

Figures 37 and 38 show that the AB Annealing partition caused a reduction in the number of inter-LP arcs with a corresponding reduction in the amount of real message traffic for all LP values. For example, with 8 LPs, the number of real messages transmitted has been reduced by 85% over the random partition. However, the corresponding speedup results were mixed, with no noticeable improvement over the SBF partition for 2, 3, 4, 5, and 7 LPs. The speedup curve shows a modest improvement with 6 LPs, and a more significant improvement with 8 LPs where it peaks at 3.89. The large improvement with 8 LPs is due to a decrease in null message traffic caused by a reduction in LP output lines from 29 to 26, along with an increase in the average lookahead.

There are a number of interesting observations that can be made about the AB annealing speedup curve. First, the only factor that appears to be limiting the performance improvement for a majority of the data points is an increase in the value of H_d. This lends credibility to the assertion that the distribution of the inter-process communications is a significant contributing factor to simulation performance. On the other hand, the largest

increase in H_d occurs with 8 LPs which corresponds to the data point with the greatest improvement in simulation speedup. In this instance, the increase in H_d appears to be dominated by the reduction in H_n and H_c. This phenomenon is likely due to a failure of the cost factor H_d to accurately capture the relationship between the distribution of the inter-process communications and the resulting simulation performance.

5.2.2 Associative Memory Array. The associative memory speedup results are shown in Figure 39. All speedup calculations are in reference to the single LP case which required an average of 4,380,074 ms to complete. Figures 40 and 41 compare the partitions statistics and resulting message traffic respectively.

As shown in Figure 39, the random partitioning speedup peaks at 3.95 with 5 LPs, and declines to 3.03 with 8 LPs. Due to the large number of total simulation events in the associative memory circuit, the communications overhead for random partitioning is dominated by real message traffic for the number of LPs used in this thesis. For example, the null to real message ratio begins at 1:25 with 2 LPs, and increases to 1:2 with 8 LPs. Further increases in the number of LPs will swing the ratio the other way, and the communications overhead will be dominated by the null message overhead. This can be seen by observing the slopes of the real and null message curves in Figure 41.

The SDF partitioning speedup curve shows consistently better results than the random partition, peaking at 4.49 with 5 LPs, and decreasing to 3.53 with 8 LPs. Looking at Figure 40, it can be seen that SDF partitioning provides a noticeable improvement over random partitioning in terms of both the number of inter-LP arcs and the number of LP output lines. For example, with 8 LPs, the number of inter-LP arcs has been reduced by 61% (from 8129 to 3145), while the number of LP output lines has been reduced by 21% (from 56 to 44). However, over 50% of the remaining inter-LP arcs originate from a single LP, resulting in a value of H_d of 333.2%.

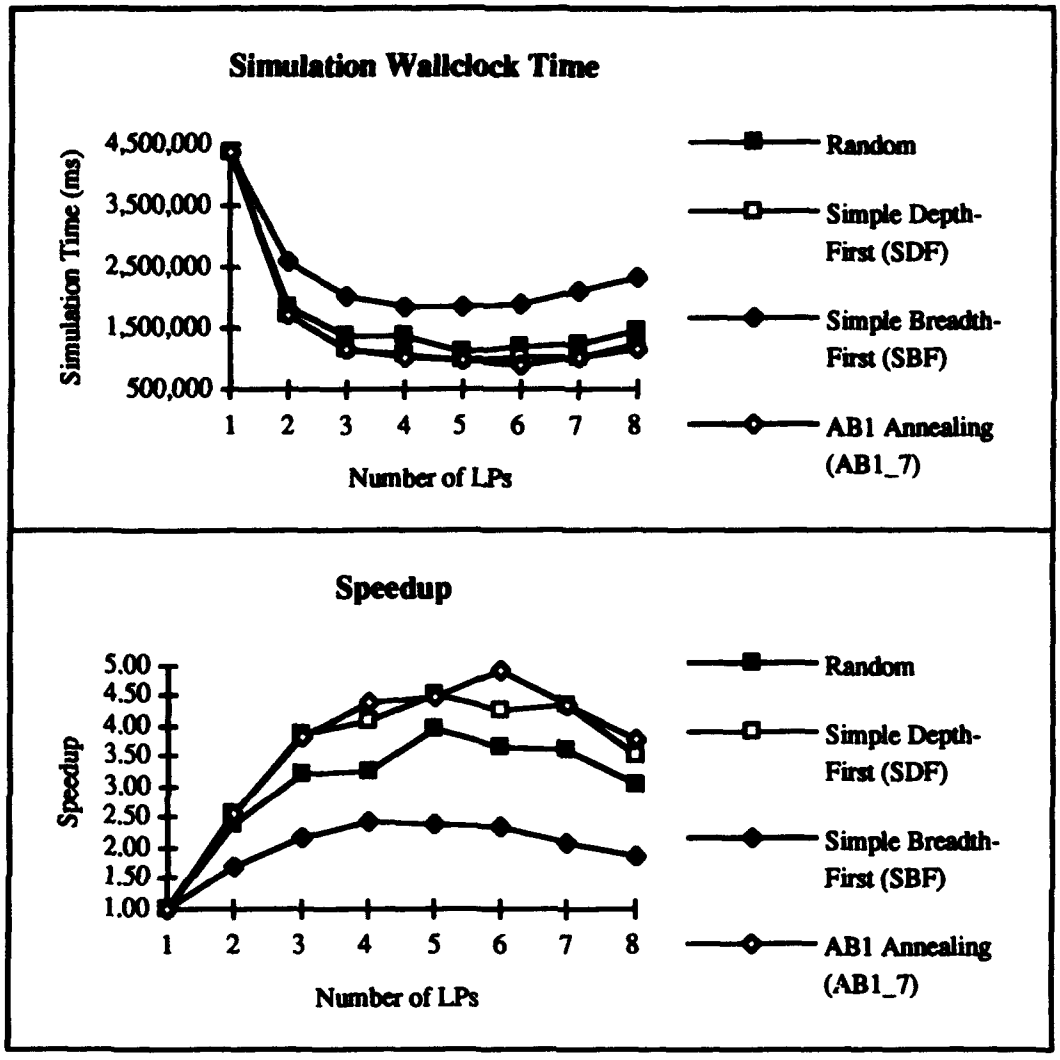


Figure 39. Associative Memory Speedup Results Comparison

Looking at Figure 41, the effect on the real message curve was even more dramatic than the speedup curve, with the number of real messages and null messages being reduced by 90% and 41% respectively in the 8 LP case. However, due to the decrease in real message traffic caused by the improved partition, the null messages begin to dominate the communications overhead at a much earlier point. The null to real message ratio with 8 LPs is 3:1 (vs. 1:2 for the random partition). Nevertheless, the total communications overhead was reduced by over 72% with 8 LPs. Although this improvement was significant, the corresponding speedup was improved by less than 17%.

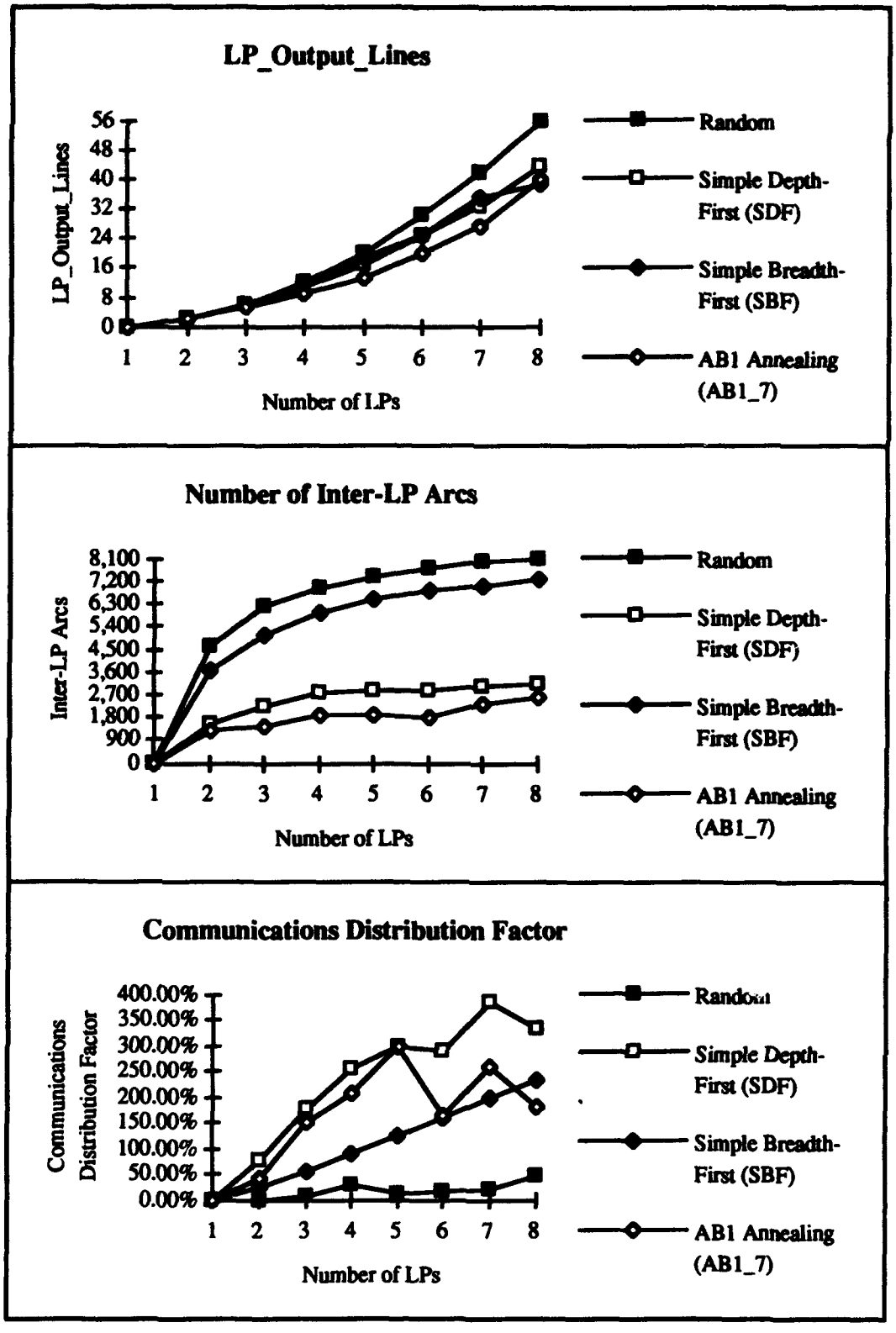


Figure 40. Associative Memory Partition Statistics Comparison

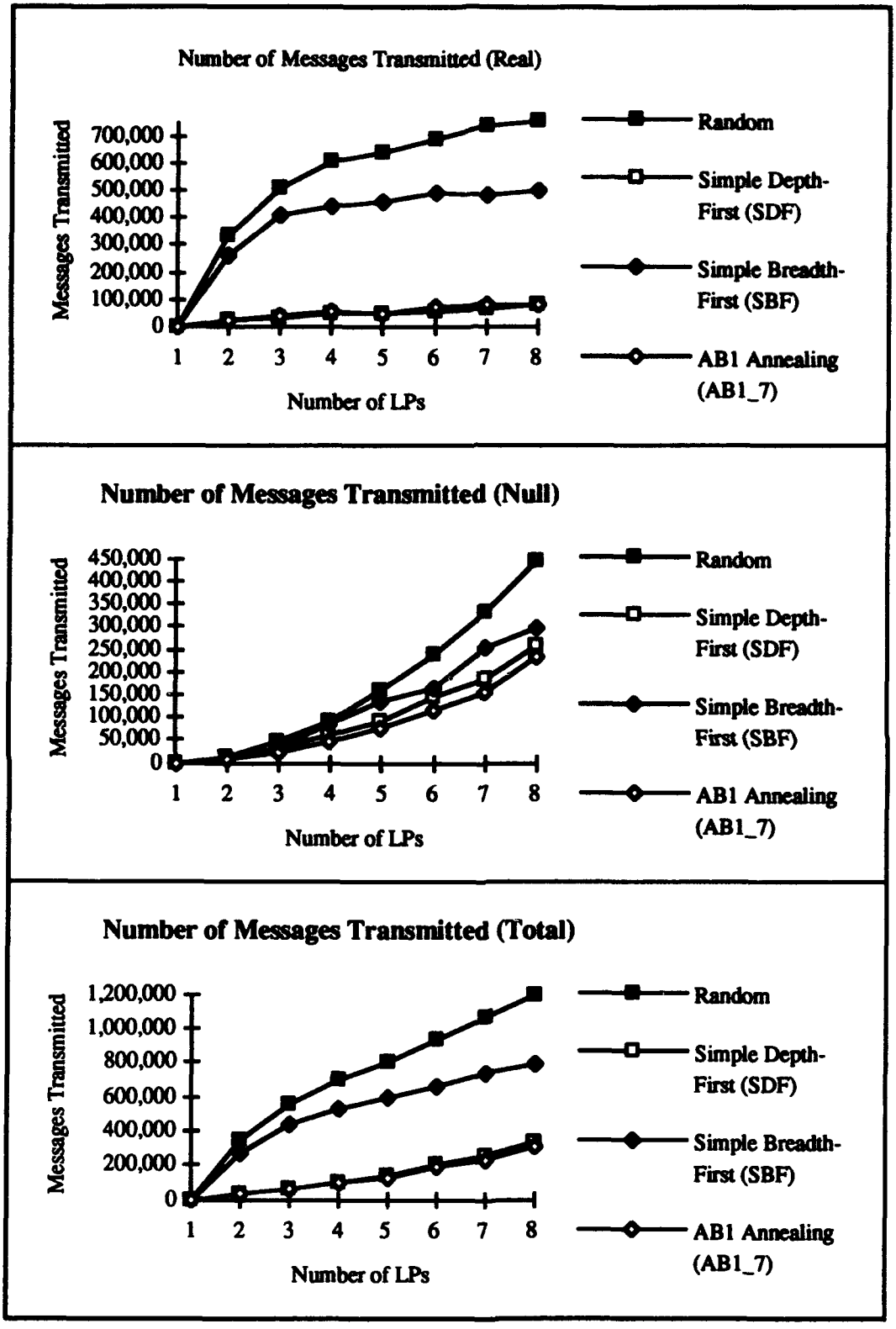


Figure 41. Associative Memory Inter-LP Message Traffic Comparison

The SBF partitioning algorithm performed very poorly on the associative memory circuit, giving speedup that was worse than random partitioning. The exact reason for the poor performance of the SBF partitions is not clear, as there were improvements in both the number of inter-LP arcs and LP output lines. However, the value of H_d was consistently higher, although it was less than that of the SDF partitions. It appears that the decrease in message overhead was not large enough to overcome the increase in H_d.

It is interesting to note in Figure 40 that the SBF and SDF partitions result in approximately the same number of LP output lines. However, Figure 41 clearly shows that the SBF partitions resulted in a noticeably higher amount of null message traffic. There are two reasons for this effect. First, the SDF partitions resulted in larger average lookahead values. This was expected since the SDF algorithm has a better chance of grouping long sequences of dependent behaviors due to the order in which the graph is traversed during partitioning. Second, when an LP sends a real message over one output line, it sends null messages over all other output lines with the same timestamp. Therefore, the large number of real messages in the SBF partitions causes an increase in the null message overhead.

For the associative memory circuit, the SDF partition was used as the initial partition to the AB annealing algorithm to get the fourth speedup curve shown in Figure 39. The following input parameters were used to the annealing algorithm:

```
Num_Iterations        - 500      Ignore_Comm_Dist_Factor - false
Max_Worthless_Iter    - 50       Topological            - false
Load_Imbal_Tol        - 0.5%     Hop_Weights            - all 1.0
```

Looking at Figure 40, it is clear that the border annealing algorithm improved the quality of the partition in terms of each of the three communications cost factors: LP output lines, inter-LP arcs, and communications distribution factor. However, the corresponding speedup results were decidedly mixed, with the new partitions performing the same as the SDF partitions with 5 and 7 LPs. However, the the highest speedup

obtained for the associative memory, 4.89, was achieved with the 6 LP AB annealing partition. Although the exact reason for the lack of improvement in the 5 and 7 LP cases is not clear, there appears to be several contributing factors.

As an example, comparison of Figures 40 and 41 shows that although the number of inter-LP arcs is reduced with the AB annealing algorithm, the number of real messages transmitted is slightly higher. This is possible because, as discussed in section 3.3.2.1, the actual real message communications load over each arc is dependent upon the signal activity of the circuit. Since this information is not available prior to simulation, each arc is assumed to have an equal cost in terms of message load.

Although real message traffic was increased slightly, the null message traffic was lowered. As expected, this was due to a consistent reduction in the number of LP output lines caused by the AB border annealing algorithm. However, the reduction in null messages is not as large as might be expected from the reduction in LP output lines. As with the SBF partitions, this is due to the slight increase in real message traffic. The net result is a modest decrease in total message traffic over the SDF partitions, with a maximum decrease of 8.67% on 7 LPs.

As stated previously, the AB annealing speedup results were mixed. The biggest increase over the SDF partitions, 15%, occurred with 6 LPs. It is interesting to note that this corresponds with the biggest drop in the communications distribution factor H_d (see Figure 40). While this is consistent with the expected results from the objective cost function, it is also interesting to note that the smallest speedup gains correspond to the test cases with the largest decrease in total message traffic (5 & 7 LPs). These inconsistent results provide a strong indication that other factors are influencing the simulation performance.

As mentioned previously, one potential source of these inconsistencies is a failure by the partition cost function to capture the true relationship between the distribution of the

communications and the simulation performance. Another possible source of the inconsistency is the artificial feedback imposed upon the simulation when the behavior graph is mapped onto the processor graph. Although this feedback is not addressed directly in this thesis, partitioning the circuit to reduce the number of LP output lines and inter-LP arcs also reduces the amount of imposed feedback. Intuitively, however, the true effect of these imposed feedback loops depends upon the behaviors involved and the signal activity of the circuit. Further research is needed in order to account for this overhead in the partition cost model.

5.3 Speedup Prediction

As discussed in section 3.3.4.2, one of the objectives of this thesis was to quantify the relationship between the quality of a partition and the resulting speedup. Although the ability to predict the speedup from the partition information may be useful, the primary purpose of this objective is to validate the partition cost function developed in this thesis. The coefficient value ß was arbitrarily set to 1.0, and the coefficient value α was selected to provide the desired relative weightings to the load imbalance and communications portions of the objective cost function. The first step in selecting α was to examine the expected magnitudes of the communications cost sub-function ($H_n H_c (1 + H_d)$) and the load imbalance sub-function (H_b) for a typical example (e.g., the wallace tree multiplier). Because of the chosen methodology selected for representing the communications cost factors, the communications cost sub-function is likely to produce a much larger value than the load imbalance sub-function. Therefore, it was desired to make α larger than ß in order to account for this difference in representation. An α value of 100.0 was chosen. This selection was validated by comparing the resulting expected speedup values against the actual speedup curves using an arbitrary value for γ. It should be noted that the relatively small load imbalances used in this thesis (*1.5% max*) *would tend to obscure any*

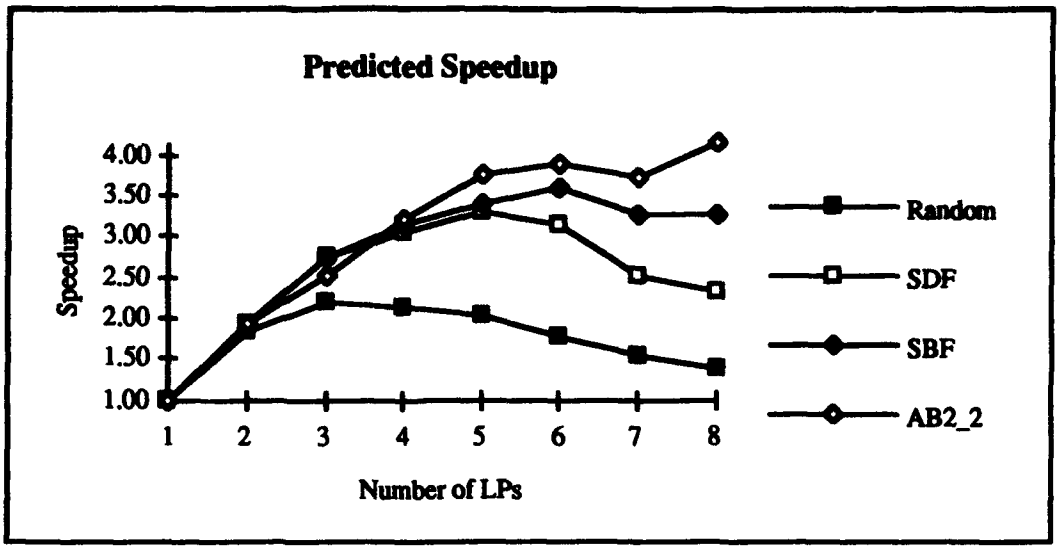


Figure 42. Wallace Tree Speedup Prediction Curves

errors with this relative weighting by causing the cost function to always be dominated by the communications cost sub-function. *Further research is required in order to find the correct relative weightings.* This research should include a re-examination of the "best" way to represent the communications cost factors[16].

The last remaining coefficient, γ, was determined to be circuit dependent. Specifically, it appears to be inversely proportional to the total number of events in the simulation. The value for γ was selected separately for each circuit by using trial-and-error to find the value which gave the best match between the predicted vs. actual speedup curves for random partitioning. Once selected, the same value was used for each of the other partitioning algorithms. All three coefficient values are given to GP-Tool, which calculates the predicted speedup as part of the partition statistics output file.

In general, the speedup prediction results were correct (but not exact) for a clear majority of the partition - LP combinations. Intuitively, this seems to indicate that the objective cost function proposed in this thesis is able to successfully model the

[16] For example, under the current implementation, H_C and H_d are calculated as percentages, but H_n is not.

dominating partition cost factors in most circumstances. Further research would allow this cost model to be refined to provide even better results.

5.3.1 Wallace-Tree Multiplier. The predicted speedup curves for the wallace tree multiplier are shown in Figure 42. For this circuit, a γ value of 0.09 was used. Although the predicted speedup values were not exact, comparison with Figure 36 shows that the partition cost model successfully predicted the correct ordering of the four partition types. For example, it correctly predicted that the SBF partitions would perform better than the SDF partitions which would, in turn, perform better than the random partitions. Except for the 7 LP case, it also correctly predicted that the AB annealing partitions would outperform the SBF partitions.

5.3.2 Associative Memory Array. The predicted speedup curves for the associative memory array are shown in Figure 43. For this circuit, a γ value of 0.025 was used. Again, the predicted speedup values were not exact, but comparison with Figure 39 shows that the partition cost model successfully predicted the correct ordering of the four partition types. For example, it successfully predicted that the SBF partition would consistently perform worse than the random partition, and that the SDF partition would consistently perform better than the random partition. However, it failed to predict the anomalous 5 and 7 LP cases where the AB annealing partitions performed no better than the SDF partitions. This further supports the assertion that under some circumstances, there are factors which contribute to the simulation performance other than those that are captured by the objective cost function model.

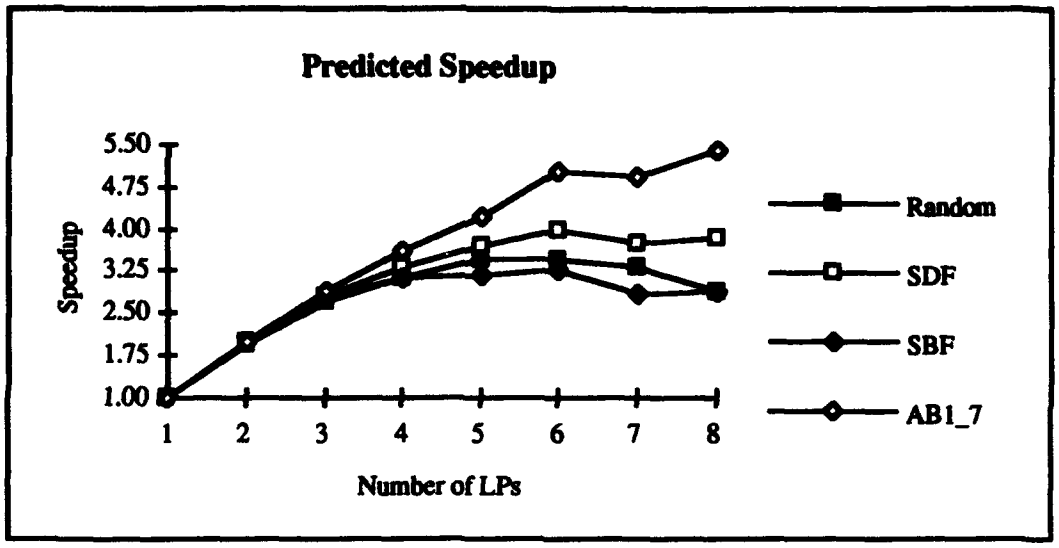


Figure 43. Associative Memory Speedup Prediction Curves

5.4 Message Traffic Analysis

This section looks more closely at the inter-LP message traffic overhead for a few representative test cases. Three types of graphs are presented: *real messages transmitted* from each LP vs. null messages transmitted from each LP; total messages transmitted from each LP vs. total inter-LP arcs originating from each LP; and total messages transmitted from each LP vs. total LP output lines originating from each LP. In each graph, all four partition types are compared. Similar graphs for additional test cases are included in Appendix D.

5.4.1 Wallace-Tree Multiplier. Figure 44 shows the real vs. null message graph for the 4 LP case, while Figure 45 shows the same graph for the 8 LP case. Clearly, for the 4 LP random partition, the communications overhead is nearly evenly divided between real messages and null messages. In addition, the communications are evenly distributed, with each LP generating a relatively equal number of messages. Although the total number of null messages is reduced for each successive partition (SDF, SBF, and AB annealing

respectively), the total number of real messages is also reduced. The resulting effect is that the inter-LP communications for each LP are dominated by the null message overhead. Figure 45 shows an identical set of relationships for the 8 LP case, except that the null messages dominate the communications overhead for the random partition as well. Intuitively, further reductions in real messages without significant reductions in null messages will have limited impact on the total communications overhead.

An additional observation from Figures 44 and 45 is that for the deliberate partitioning strategies of SDF, SBF, and AB annealing, the remaining message traffic is no longer evenly distributed among all of the LPs. Observation of this phenomenon lead to the addition of the communications distribution factor (H_d) to the objective cost function as discussed in section 3.3.2.2. Figure 46 attempts to validate the communications distribution factor by showing the relationship between the total number of messages transmitted from each LP and the total number of inter-LP arcs originating from each LP.

In general, the results are decidedly mixed. There appears to be a detectable relationship between output arcs and messages transmitted for the random and SBF partitions, but not the SDF or AB annealing partitions. Additional examples with similarly mixed results are included in Appendix D. Collectively, this data supports the assertion that the distribution of the communications load may be a factor in the simulation performance, but the cost function proposed in this thesis fails to accurately model the communications bottleneck in some instances.

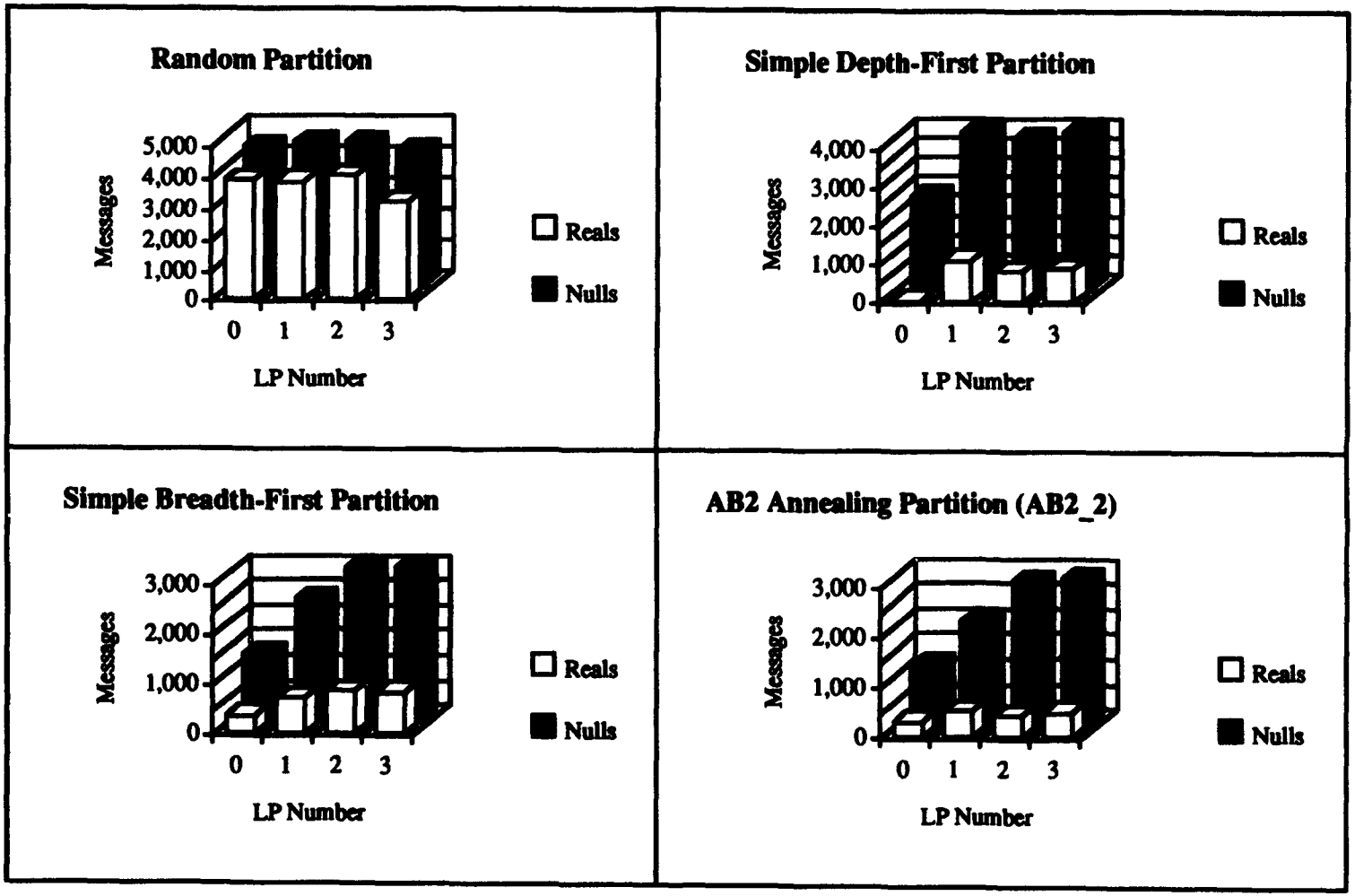


Figure 44. Wallace Tree 4 LP Reals Sent vs. Nulls Sent Message Analysis

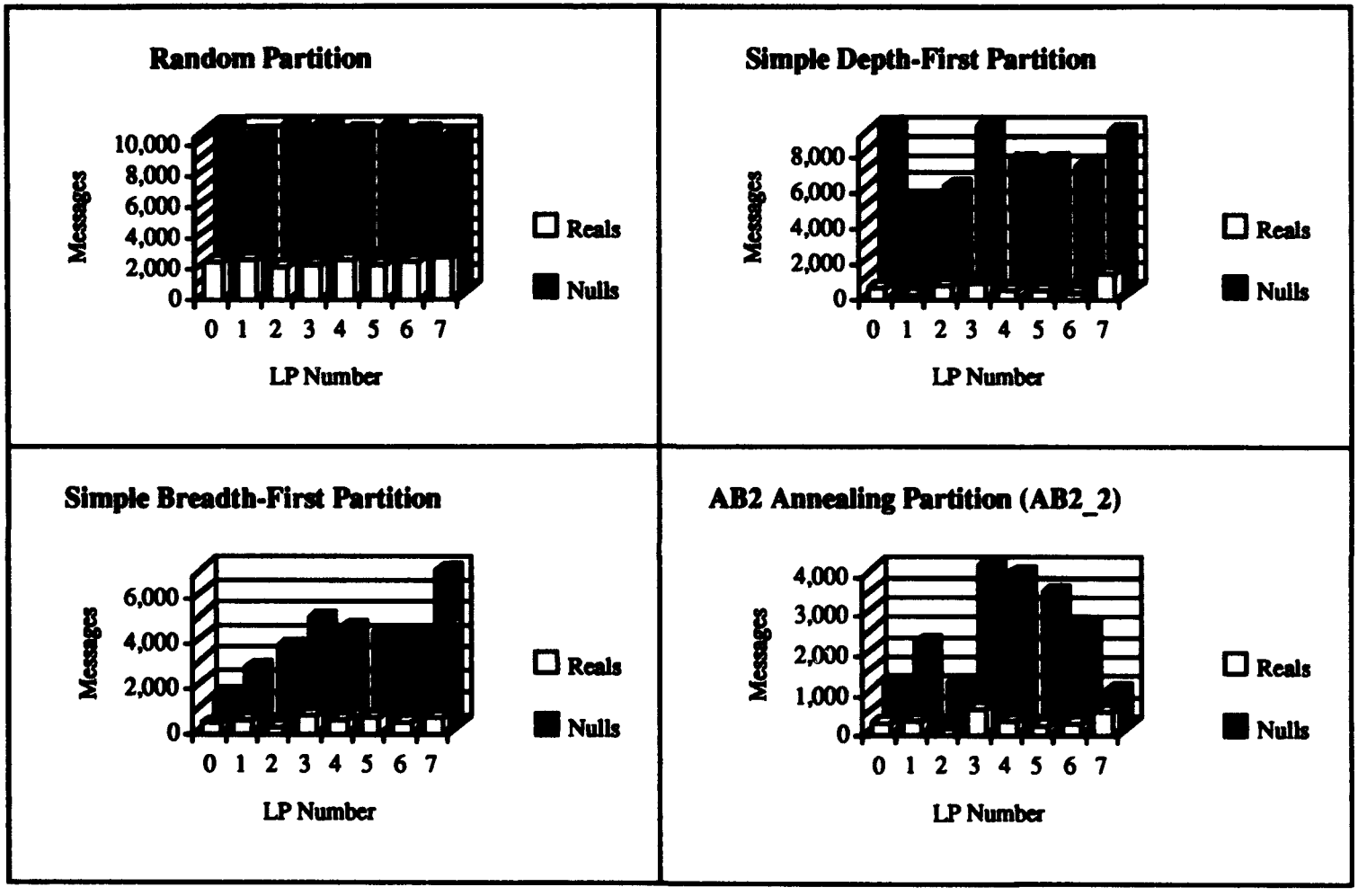


Figure 45. Wallace Tree 8 LP Reals Sent vs. Nulls Sent Message Analysis

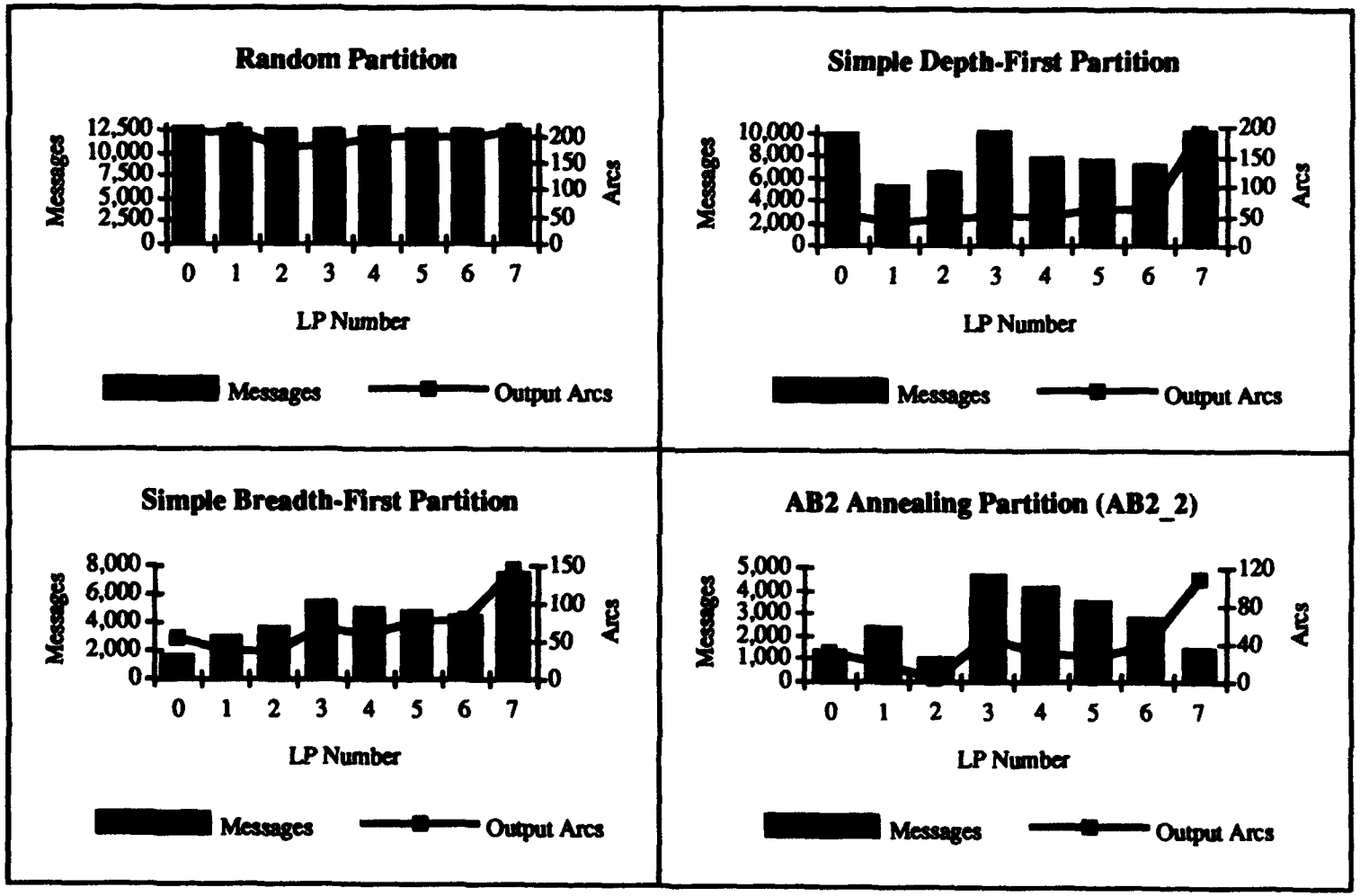


Figure 46. Wallace Tree 8 LP Total Messages Sent vs. Output Arcs

One attempt at improving the method of modeling the distribution of the inter-LP communications involved the relationship between the total messages transmitted from each LP and the corresponding number of output lines for that LP. This relationship was investigated because of the fact that for relatively large LP values, the null messages dominate the total message counts, and the number of LP output lines are the major contributor to the number of null messages. Figure 47 shows this relationship for the wallace tree 8 LP case. Clearly, there is a detectable relationship for the majority of the partition types. A single exception is LP7 for the AB annealing partition. This apparent anomaly is explainable, however, by the fact that LP7 has no input arcs. With no input arcs, it never has to block for input. Because it never blocks, it never sends blocking nulls. Additional examples with similar results are included in Appendix D.

However, despite these positive results, revising the communications distribution factor H_d to be based upon the number of LP output lines rather than the number of output arcs resulted in a degradation of the speedup prediction curve results. Nevertheless, these results indicate that this relationship deserves further investigation.

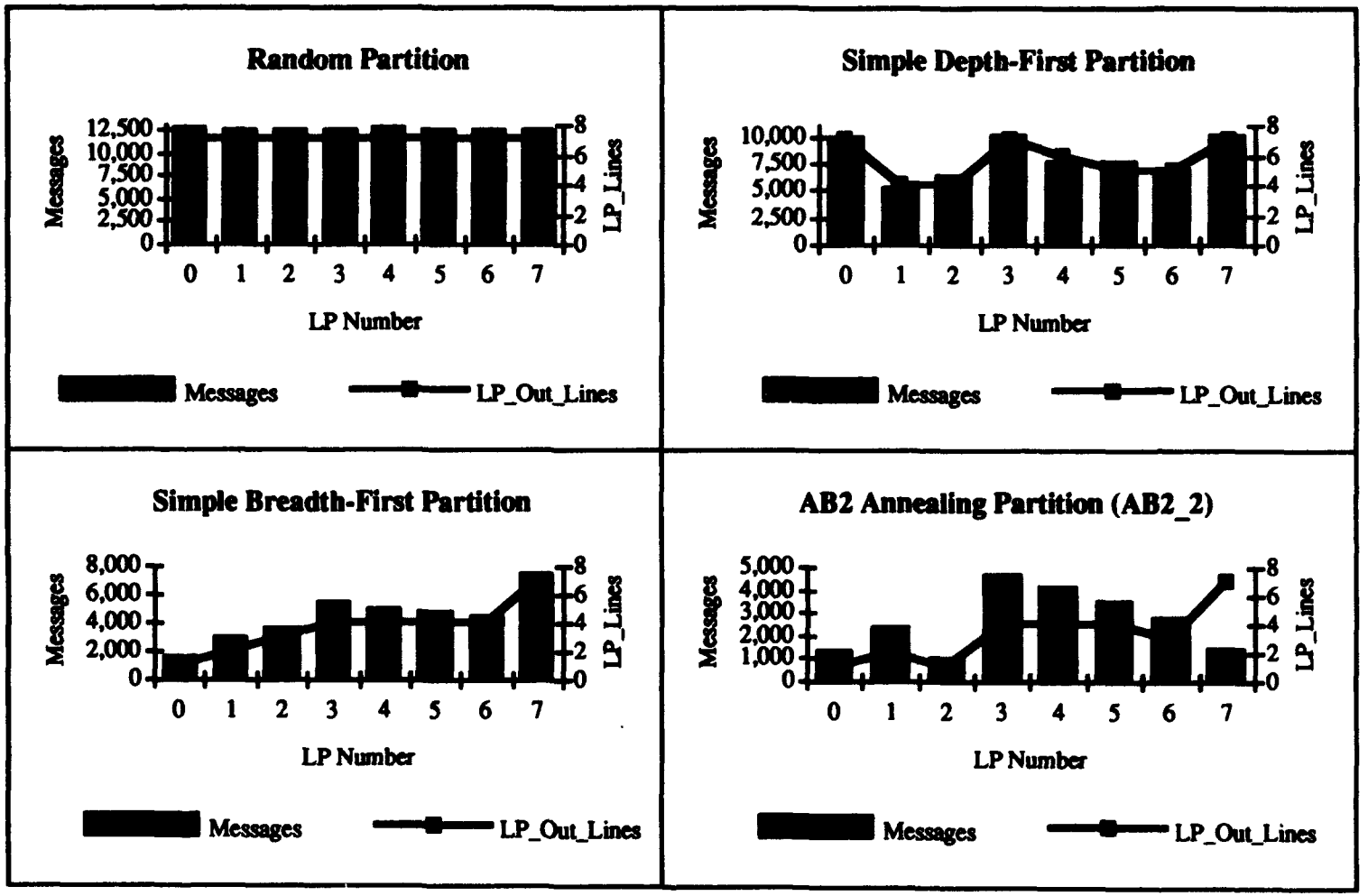


Figure 47. Wallace Tree 8 LP Total Messages Sent vs. LP Output Lines

5.4.2 Associative Memory Array. A similar set of graphs is presented for the associative memory array in Figures 48 through 51. Specifically, Figure 48 shows the real vs. null message graph for the 8 LP case. Notice that for the 8 LP random partition, real messages continue to dominate the total inter-LP message overhead. However, as mentioned in section 5.2.2, this situation will reverse itself as the number of LPs is increased further. Another interesting item from this figure is the results of the SBF partition. Notice that for a few LPs, the communications overhead continues to be dominated by real messages. Furthermore, the number of real messages transmitted by LP5 and LP7 each exceed the average number of real messages transmitted by all LPs in the random partition by as much as 109%. This communications bottleneck is the most likely culprit in the poor performance of the SBF partitions for the associative memory array.

As stated previously, the 6 LP AB annealing partition provided the maximum speedup for the associative memory circuit. Figures 49 through 51 show the 6 LP case of the real vs. null messages graph, the total messages vs. output arcs graph, and the total messages vs. LP output lines graph respectively. The results are similar to those for the wallace tree. An exception is the total messages transmitted vs. the LP output lines for the SBF partition in Figure 51. In this particular instance, there is no discernible relationship between the total number of messages transmitted by each LP and the number of output lines originating from each LP. Additional examples with similar results are included in Appendix D.

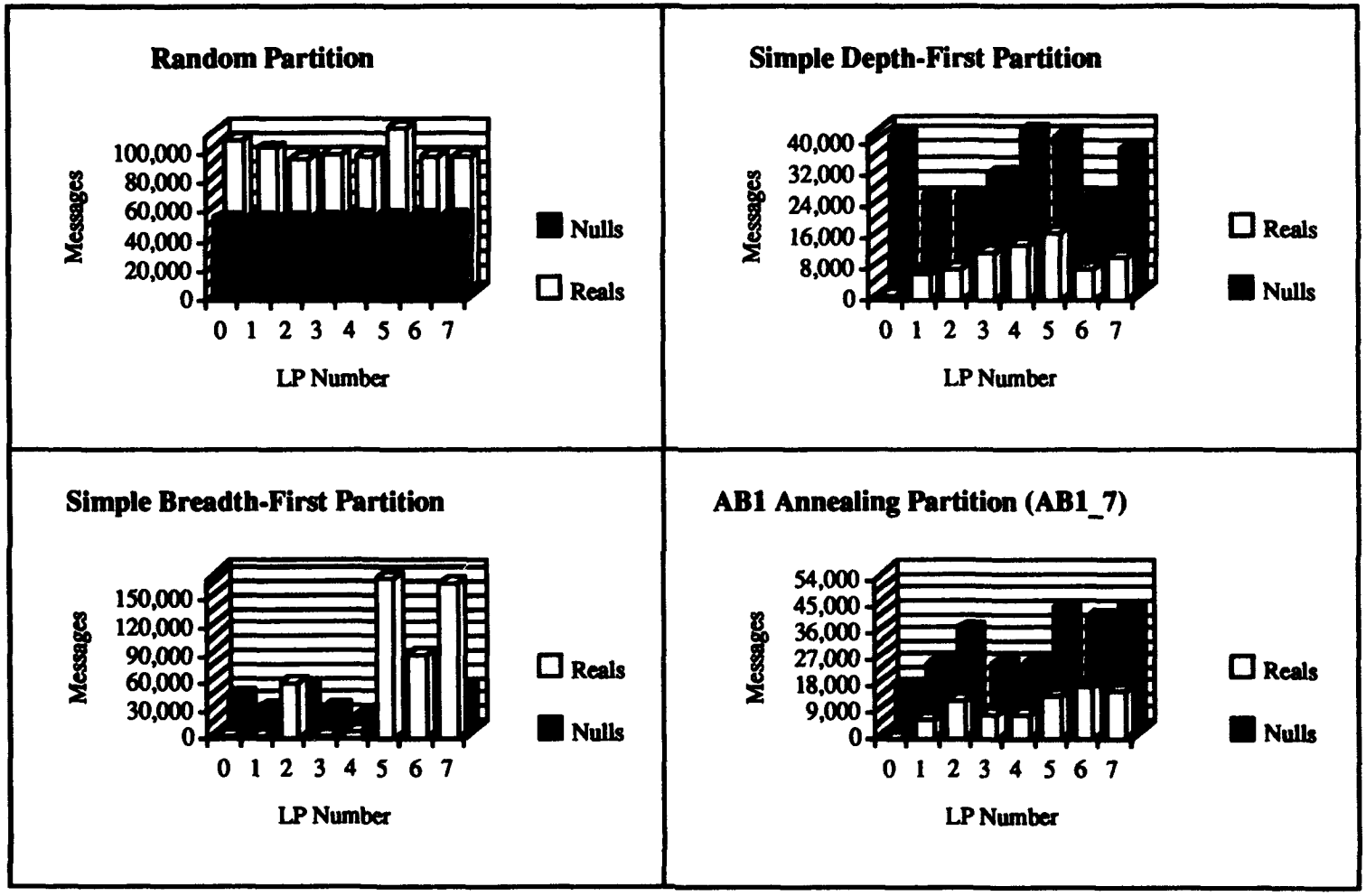


Figure 48. Associative Memory 8 LP Reals Sent vs. Nulls Sent Message Analysis

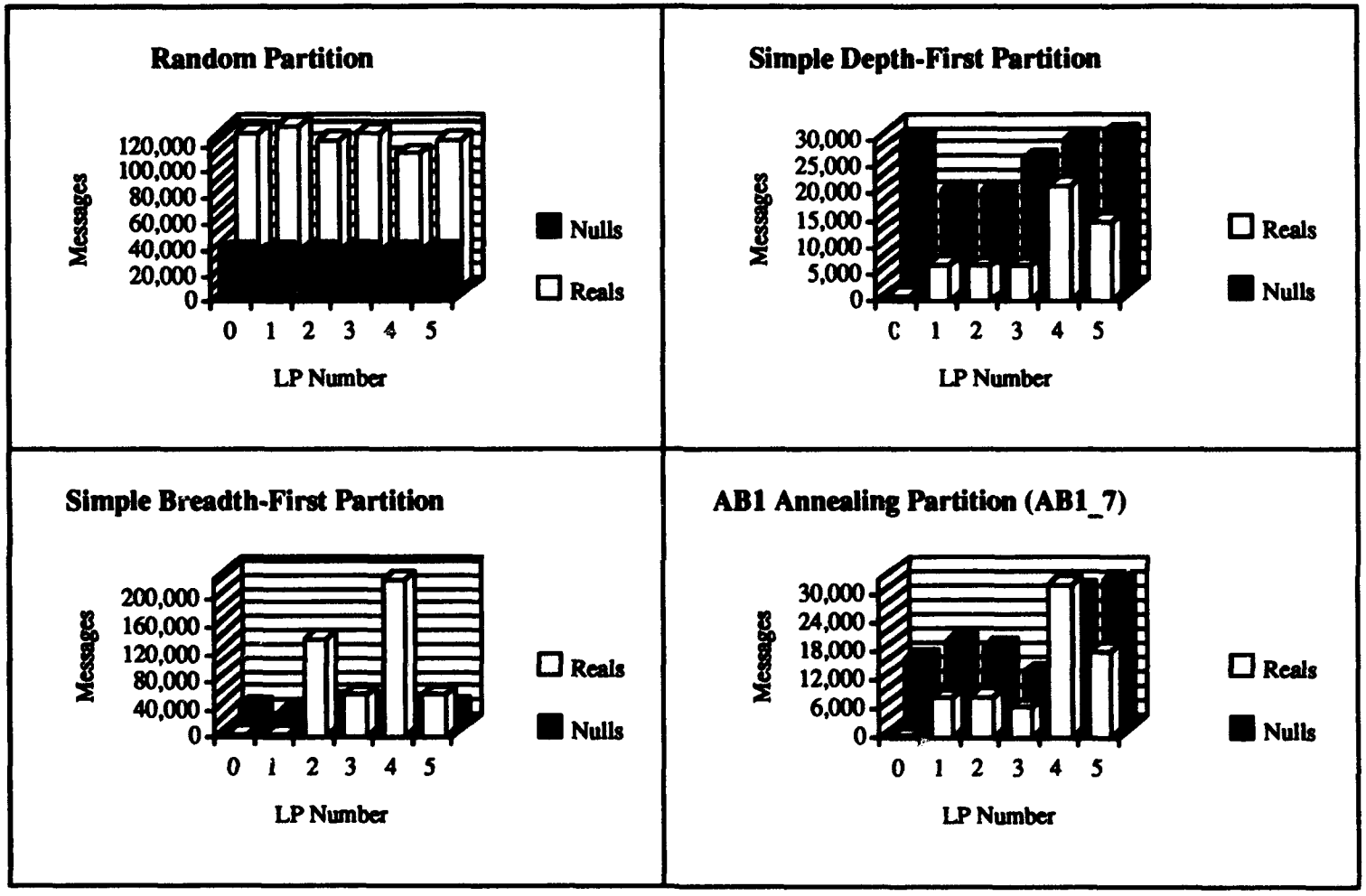


Figure 49. Associative Memory 6 LP Reals Sent vs. Nulls Sent Message Analysis

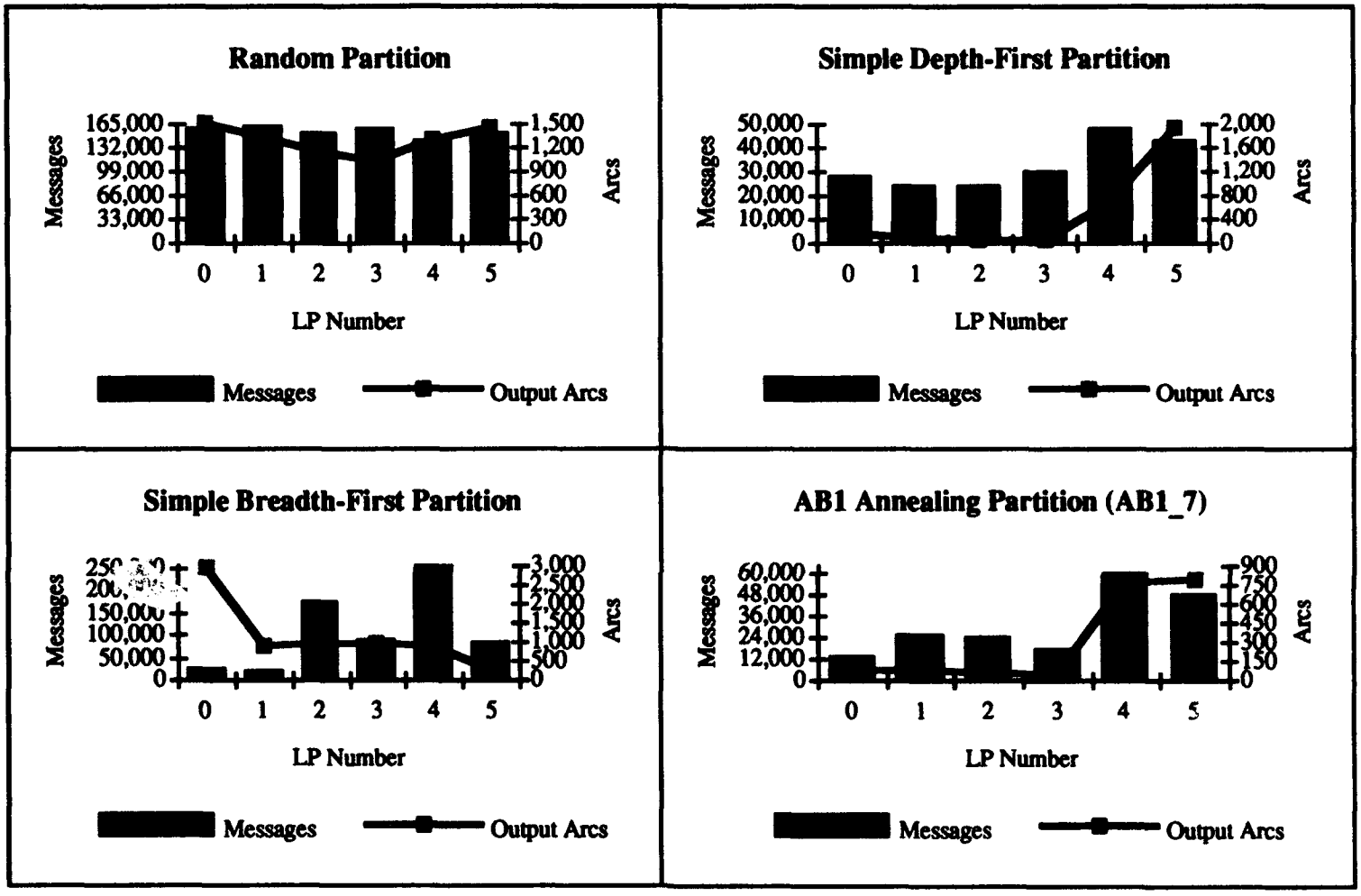


Figure 50. Associative Memory 6 LP Total Messages Sent vs. Output Arcs

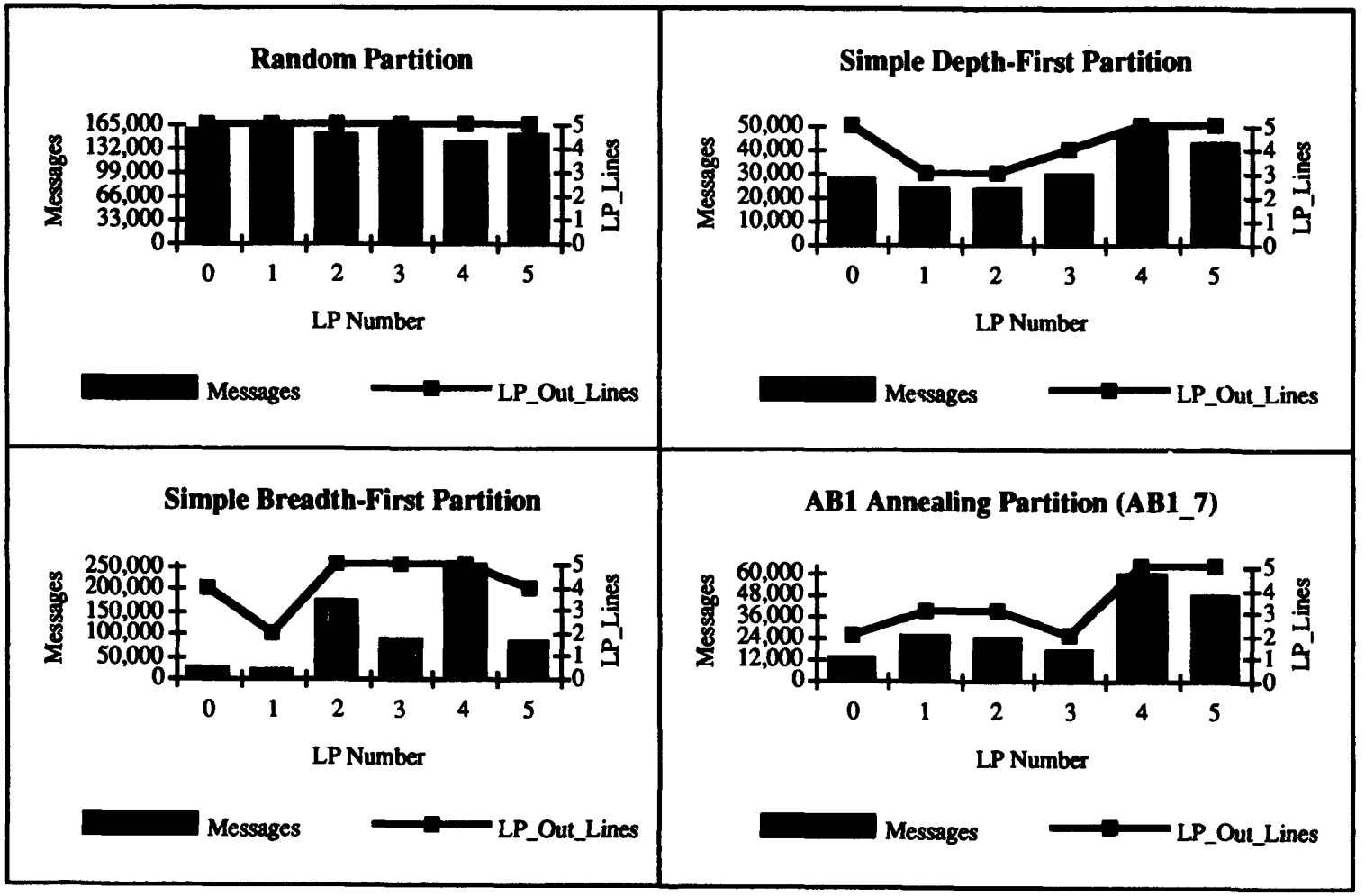


Figure 51. Associative Memory 6 LP Total Messages Sent vs. LP Output Lines

Table 2. Predicted vs. Actual Effect of Increased Lookahead

Number of LPs	Expected Nulls	Actual Nulls	% Difference
2	1,964	1,985	-1.06%
3	4,796	5,733	-16.34%
4	9,809	10,180	-3.64%
5	14,499	15,102	-3.99%
6	16,980	19,949	-14.88%
7	26,200	27,118	-3.39%
8	28,550	30,854	-7.47%

5.4.3 Increasing Lookahead. Sections 3.3.2.3 and 3.3.2.4 discuss the relationship between the average lookahead in the `lpx.arcs` file, the null message overhead, and overall simulation performance. This section uses the wallace tree SBF partitions to present a quantitative example to illustrate these relationships. To make the comparison, the SBF `lpx.arcs` files were modified to contain the wallace tree normal lookahead value of 2 ns for all LP output lines. The simulations were then re-run for comparison with the original SBF results with the increased lookahead.

In section 3.3.2.4, an assumption is made that the logical delay value for an LP output line will only be used to determine the timestamp of a null message approximately 50% of the time. This assumption is then used as the basis for a modified calculation of L_{arcs}, which estimates the impact of the average lookahead value on the null message overhead. Table 2 presents data for the wallace tree SBF partitions to demonstrate the validity of this assumption. Specifically, it uses the modified equation for L_{arcs} to calculate the expected number of null messages and compares it against the actual number of null messages. The value for expected null messages is calculated by multiplying the value of L_{arcs} by the number of null messages transmitted when all logical delays in the `lpx.arcs` file were set to their normal value (2 ns). As seen from the table, the actual number of null messages was within 10% of the expected value in a majority of the cases, and was within 20% in all cases.

Figure 52 shows the comparison between the SBF partitions with the naturally increased lookahead and the SBF partitions in which all lookahead values have been set to the normal value of 2 ns. Specifically, it shows the comparison in speedup, average lookahead, and null message traffic. As can be seen from the figure, the increase in lookahead had a consistent effect of decreasing the null message overhead and increasing the speedup. However, the net effect on speedup was essentially negligible, and the number of null messages is still proportional to the number of LP output lines. Therefore, while increasing the average lookahead values will help to optimize the simulation performance, it does not appear to be a potential source of significant speedup gains.

5.4.3.1 Calculating Lookahead. While increasing the average lookahead reduces the null message overhead, calculating the correct lookahead values is a potential computational bottleneck. The maximum lookahead value for an LP output line is defined as the minimum path from all inter-behavior arcs entering the LP (and all source behaviors in the LP) to all inter-behavior arcs exiting the LP that correspond to the given LP output line. The minimum path is defined as the sum of the logical delays of the behaviors on the path.

The current algorithm used by GP-Tool is a recursive algorithm that begins at each input arc to the current LP (LP_a) and traverses all possible paths through the LP until it reaches an external LP, tracking the length of the current path (in terms of logical behavior delays) along the way. When the current path reaches an external LP (LP_b), the minimum path on record from LP_a to LP_b is compared to the length of the current path and updated if necessary. The process is repeated for each input arc and each source behavior for each LP. This algorithm has proven to be the most efficient method of calculating the lookahead values in most situations. Two exceptions are discussed in the next section.

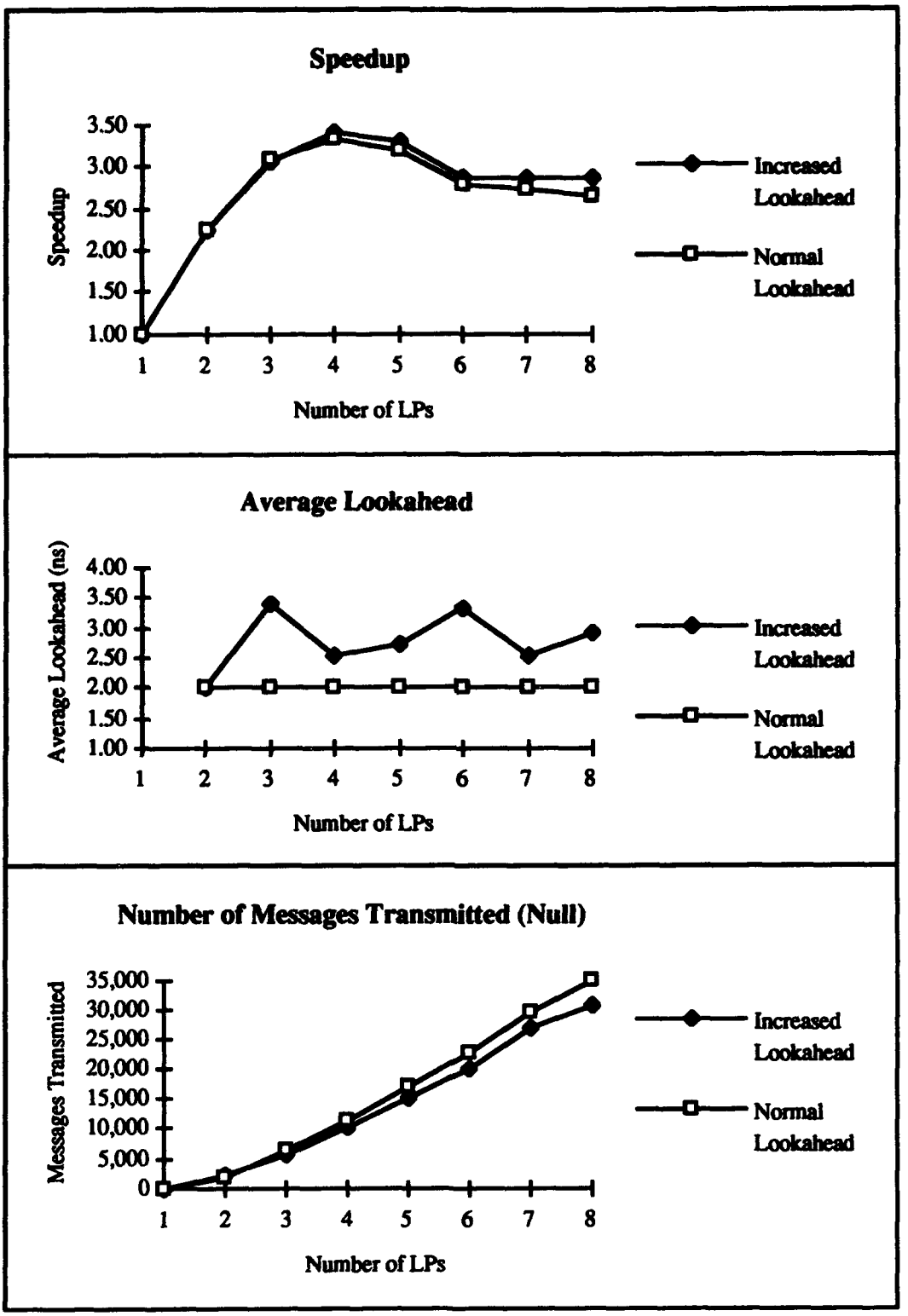


Figure 52. Effect of Increased Lookahead on Wallace Tree SBF Partitions

5.4.3.2 Lookahead Anomalies. This first anomaly related to the calculation of the average lookahead values for the `lpx.arcs` files deals with the running time of the recursive .arcs routine discussed in the previous section. Under certain circumstances, this algorithm can be extremely inefficient. Specifically, calculating the lookahead values for the wallace tree circuit with 2-4 LPs can take 30 minutes or more. This occurs because the SDF partitioning algorithm results in partitions in which the first LP contains relatively long paths of dependent behaviors. This, in turn, increases the number of paths through the LP as well as increasing the level of recursion required by the algorithm.

Comparison with other algorithms, however, indicates that the increased computation time is out of proportion to the longer paths caused by the SDF partition. For example, Dijkstra's shortest path algorithm, which finds the shortest path from a given behavior to all other behaviors, runs much faster than the anomolous recursive case. The problem was not experienced on the associative memory circuit, which is more than 4 times larger than the wallace tree multiplier. Furthermore, as the number of LPs was increased, the computation bottleneck for the wallace tree SDF partitions decreased.

An alternative method for calculating the correct lookahead values based upon Dijkstra's shortest path algorithm was implemented as well (although it is not in the current version of GP-Tool). For each input arc and source behavior in LP_a, this approach uses Dijkstra's algorithm to calculate the shortest path to all other behaviors in the system. Not all of these paths are relevant to the lookahead value. For example, the path from behavior i in LP_a to behavior j in LP_c is of no interest if there is no direct connection from LP_a to LP_c. Therefore, this algorithm involves significant extraneous computations. Although this algorithm is independent of the number of LPs or the quality of the partition, an additional disadvantage is that the algorithm is proportional to N(N-1), where N is the number of behaviors in the system. In general, this algorithm was less

efficient than the recursive algorithm, except for those few cases in which the recursive algorithm experienced the anomalous computational bottlenecks.

The second anomaly related to the calculation of the lookahead values for the `lpx.arcs` files involved the associative memory 8 LP SBF partition. Specifically, the recursive algorithm produced lookahead values which caused message out-of-order errors during simulation. This occured when the lookahead values in the `lpx.arcs` file were too large, causing the transmission of null messages with timestamps that were too large. Theoretically, calculating the lookahead values as described in the previous section should prevent this from occurring. Analysis has failed to find any errors in the algorithm, and the problem was not observed on any other circuit/partition/LP-value combination. It was manually resolved by reducing the lookahead values in the `lpx.arcs` file in small increments until the out-of-order errors disappeared.

5.5 AB Border Annealing Algorithm

One of the primary objectives of this thesis research was to make the partitioning strategy implemented efficient in terms of required computation time. As discussed in chapter 4, the annealing algorithm continues until a maximum number of iterations have been executed, or until a specified number of consecutive iterations have been executed with no net improvement in the cost function. Under the current implementation of GP-Tool, there is no option for setting the tolerance for measuring changes in the cost function. Rather, changes are measured to the precision provided by the standard floating point data type used in the Ada compiler. As a result, the annealing process will continue as long as minor improvements are being made.

For example, Figure 53 shows the partition statistics for the 8 LP wallace tree AB2 annealing partition as they vary over the iterations of the AB border annealing algorithm. Recall that in this partition, the option `Ignore_Comm_Dist_Factor` was set to true. Thus,

H_d is allowed to increase in an effort to maximize the reductions in H_c and H_n. Although the algorithm executes 168 iterations before terminating, it is clear that minimal improvement was made after the first 35 iterations. This type of behavior, in which the majority of the improvement was made in the first few iterations, is typical for the test cases performed to date.

The AB border annealing algorithm has not been instrumented to allow for detailed timing measurements. However, the entire AB annealing process for this test case took approximately 59 sec to complete, for an average of 0.35 sec per iteration. It should be noted that not all iterations will require the same amount of computational time. For example, as the state of the partition is improved through successive iterations, fewer behaviors will be queued for reassignment consideration. Thus, as the algorithm progresses, the time per iteration will show a decreasing trend.

As a point of comparison, rough measurements were taken on the time required to produce a random partition[17] and an SDF partition for both the wallace tree and the associative memory with 8 LPs. For the wallace tree, the random partition took approximately 7 secs, while the SDF partition took less than 1 sec. For the associative memory, the random partition took approximately 71 secs, while the SDF partition took less than 3 secs.

5.6 Increasing the Number of Processors

The results presented above were limited to an 8-node iPSC/2 hypercube. In order to validate the results on a larger number of processors, the wallace tree circuit was run on an iPSC/860 using up to 32 nodes. The simulation speedup results, partition statistics, and inter-LP message counts are presented in Figures 54 to 56 respectively.

[17] Note that the random partitioning algorithm was not written for maximum efficiency.

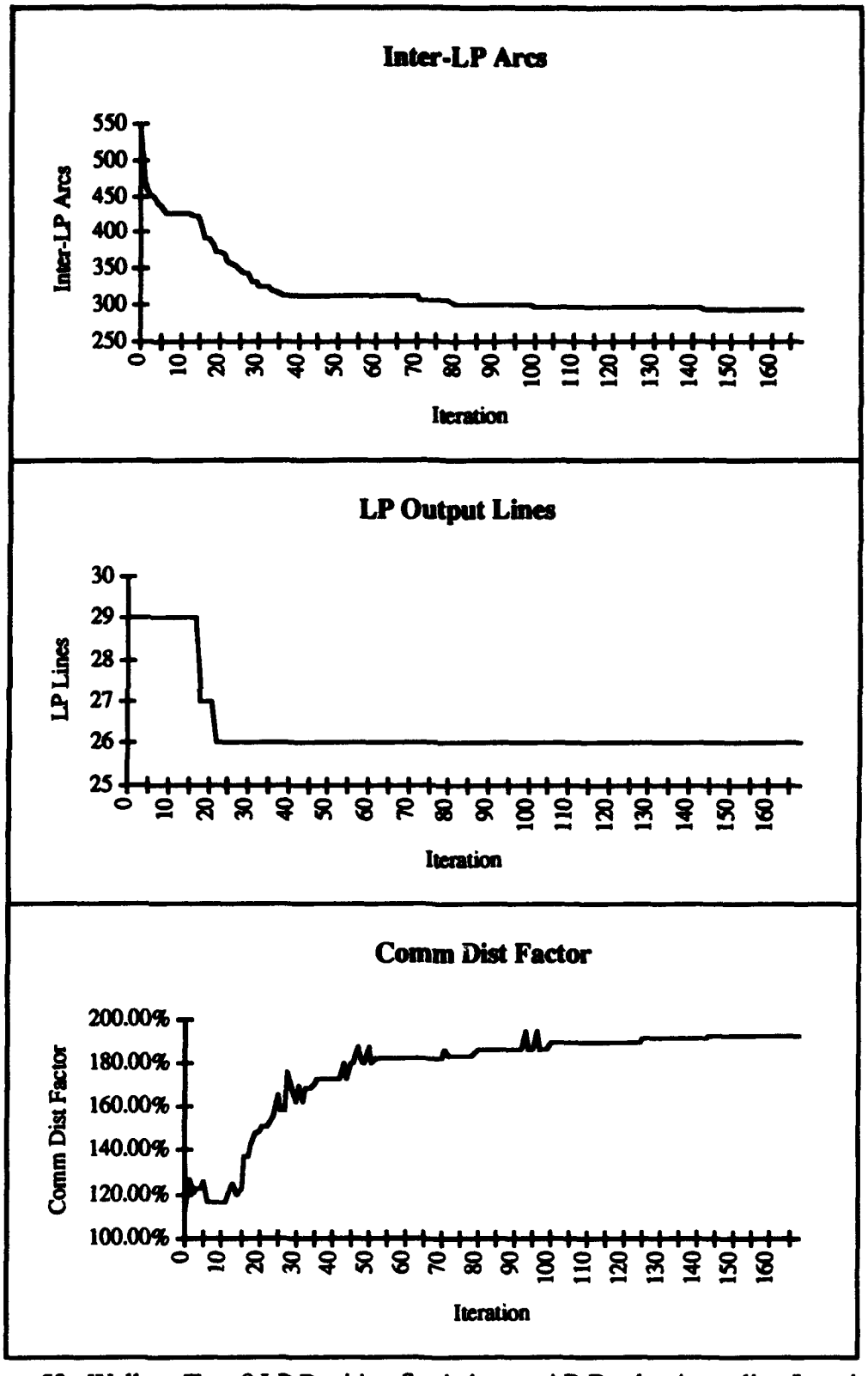


Figure 53. Wallace Tree 8 LP Partition Statistics vs. AB Border Annealing Iterations

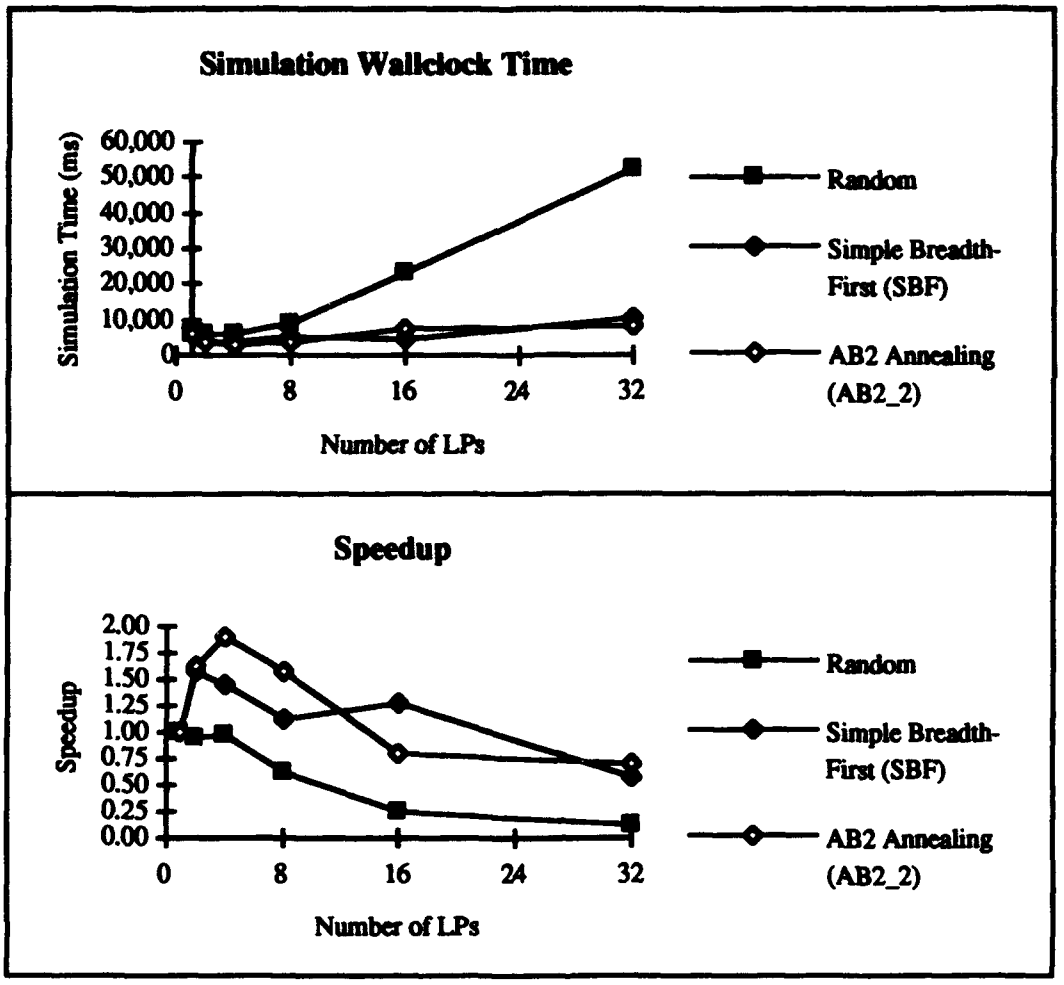


Figure 54. iPSC/860 Wallace Tree Speedup Results Comparison

As can be seen from Figure 54, the increased processing power of the i860 processors provides an order of magnitude speedup over the iPSC/2 for the single LP case. Because single LP case runs so much faster on the i860, it is much more difficult to obtain speedup with multiple processors. For example, the speedup for the random partition was less than 1.0 for all LP numbers greater than 1, and the maximum speedup obtained was 1.9 on 4 LPs for the AB annealing partition. Despite these differences, the patterns relating the partition statistics to the inter-LP message traffic and speedup are identical to those for the iPSC/2 results.

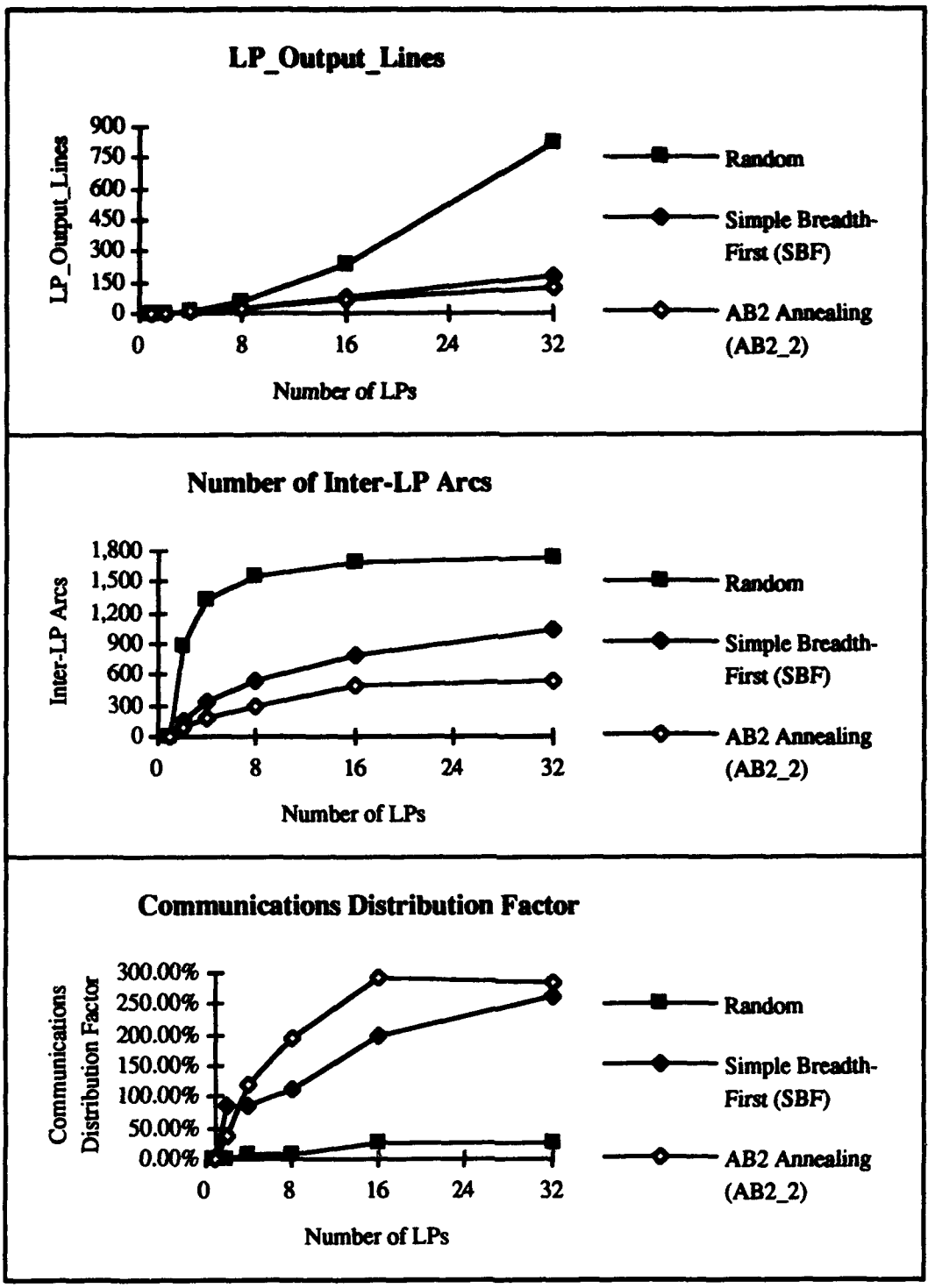


Figure 55. iPSC/860 Wallace Tree Partition Statistics Comparison

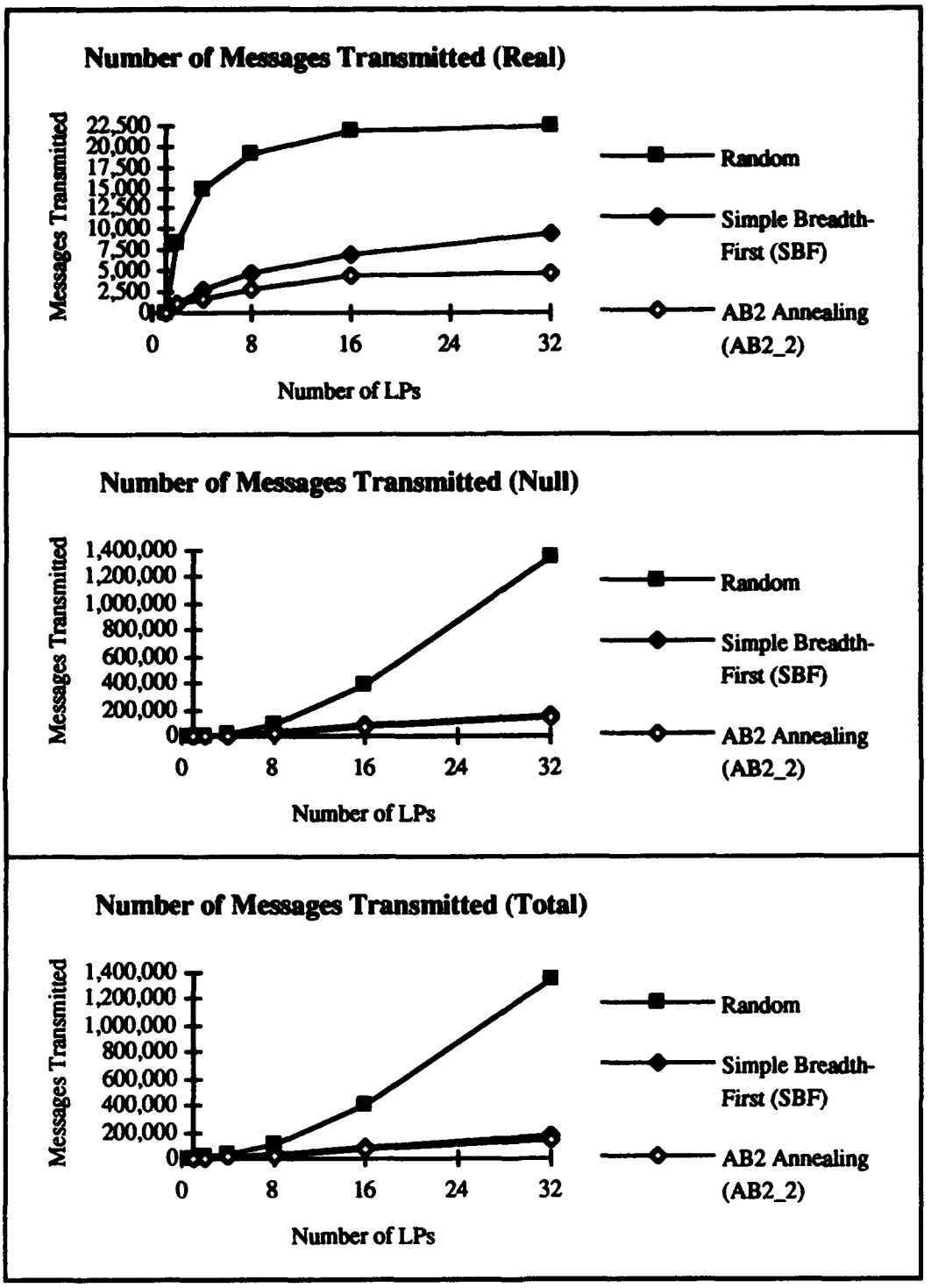


Figure 56. iPSC/860 Wallace Tree Inter-LP Message Traffic Comparison

VI. Conclusions and Recommendations

6.1 Research Summary

As modern integrated circuit designs grow larger and more complex, the time required to perform sequential VHDL simulations becomes more burdensome. In order to execute complex simulations in a reasonable amount of time, a parallel VHDL simulator should be used to simulate hierarchical structural VHDL circuits. Parallelism is introduced by partitioning the circuit behaviors among the available processors to form logical processes (LPs). Signal changes are shared among the LPs by event messages.

Parallelism by itself, however, fails to provide satisfactory speedup results due to the overhead required to communicate signal changes and maintain synchronization between LPs. The amount of overhead is directly dependent on how the circuit behaviors are partitioned among the logical processes.

In this research effort, a circuit behavior inter-dependency structure is extracted from the first iteration of VSIM's sequential simulation cycle. This information is used to build a graph representing the structure of the circuit being simulated with the circuit behaviors as vertices and their inter-dependencies as directed arcs. Using various graph traversal techniques to account for the circuit inter-dependency structure, the circuit is divided into the desired number of LPs. A border annealing algorithm is then employed to refine the quality of the partition by selectively reassigning behaviors to different LPs.

Two relatively large circuits (an 8x8 wallace tree multiplier, and a 16x16 associative memory array) have been used as subjects on which to test partitioning techniques. Speedup results are compared to those produced by a random partitioning of the circuit behaviors.

As an aid in making reassignment decisions during the border annealing process, an objective cost function is formulated in order to measure the quality of a given partition.

This cost function is designed to account for the additional communications overhead resulting from the conservative null message PDES synchronization protocol. In addition, an attempt is made to account for unevenness in the distribution of the communications overhead. Finally, an attempt is made at quantifying the relationship between the quality of a partition as measured by the objective cost function and the resulting simulation performance.

6.2 Conclusions

The following general conclusions can be made about partitioning hierarchically built structural VHDL circuit simulations:

- *Deliberate partitioning schemes improve simulation speedup.* The primary research objective of demonstrating improved speedup over random partitioning was accomplished. This research has found that, in general, a deliberate partitioning algorithm which accounts for the complex inter-dependency relationships of the circuit behaviors will tend to reduce the communications overhead and improve the simulation performance.
- *The partition cost function must account for more than load imbalance and the number of inter-LP arcs.* As stated in the research objectives, an effort was made to determine a meaningful method of measuring the *cost* of a partition. Data analysis shows that as the number of LPs is increased, the null message traffic due to the conservative PDES synchronization protocol begins to dominate the inter-processor communications overhead. Reducing the real message traffic by reducing the inter-behavior arcs which cross LP boundaries serves to enhance the dominance of the null message overhead. Not accounting for the null message overhead will give an inaccurate picture of the quality of a given partition. Therefore, *null message synchronization overhead must be accounted for in the*

partition cost. In addition, this research indicates that the distribution of the remaining communications among the LPs also impacts the performance of the simulation. Additional research is needed in order to determine the exact nature of this latter relationship.

- *Null message overhead is directly proportional to the number of arcs in the LP connectivity graph*. Results have clearly shown that the number of null messages required to maintain synchronization is directly proportional to the number of arcs in the LP connectivity graph (referred to as "LP output lines"). By decreasing the connectivity between LPs using a deliberate partitioning scheme, it is possible to reduce the null message overhead and improve simulation performance. However, it appears as though the best method for reducing null message overhead is to avoid imposed feedback among the LPs (i.e. make the LP connectivity graph acyclic).
- *Further reductions in real message overhead will have negligible impact on simulation performance*. The partitioning algorithms used in this thesis research have made significant reductions in the amount of real message communications overhead compared to a random partitioning of the circuit behaviors. The null message overhead is also reduced, but by a much smaller margin. As a result, the inter-processor communications overhead is dominated by the null message traffic for a relatively small number of LPs. The problem is exacerbated as the number of LPs is increased. Further reductions in real message traffic without significant reductions in null message traffic will have a negligible impact on the total inter-processor message traffic.
- *The proposed partition cost function provides an accurate means for comparing the relative quality of different partitions*. With few exceptions, it was shown that by relating the partition cost to the expected simulation speedup, the proposed

partition cost function could correctly predict the relative performance ordering of the various partitioning schemes used.

- *The proposed AB border annealing partitioning algorithm provides an effective means of iteratively improving a partition.* Data analysis shows a consistent reduction in the number of inter-LP arcs and LP output lines by using the AB border annealing algorithm. However, due to the continued dominance of the null message overhead, the corresponding simulation performance improvement is often insignificant.

6.3 Recommendations for Further Research

6.3.1 Circuit Partitioning Recommendations. Significant progress has been made in this thesis research towards achieving improved simulation speedup through a deliberate circuit partitioning strategy. Some suggested areas of research for expanding upon this progress are:

- *Eliminate imposed feedback among LPs.* By producing an acyclic LP connectivity graph, circular waiting among LPs will be eliminated. This will reduce the amount of LP blocking as well as the the null message overhead. The suggested methodology is to produce an acyclic initial partition (treating strong components as indivisible blocks), and modify the objective cost function so that LP feedback is not introduced during the border annealing process.
- *Continue exploring relationship between the distribution of the inter-processor communications and the simulation performance.* Data from this research has shown that an uneven distribution of the inter-processor communications may have a negative impact on simulation performance. The exact nature of this relationship is still undefined. It is possible that the elimination of feedback among LPs may negate the effects of uneven communications distribution.

6.3.2 Parallel Simulation Recommendations. There remains significant work to be done in the future in terms of the parallel VHDL simulation application, VSIM, and synchronization protocol. Some specific suggestions for further work are:

- *Implement a more optimistic PDES synchronization protocol.* While it may be possible to gain additional speedup by modifying the rules for sending null messages, it is unlikely that such gains will be significant. One recommended approach involves the use of "local rollbacks" (11:22). Under this approach, each LP maintains a safe-time as in the current null message approach, but is allowed to process events as fast as possible, potentially advancing its local simulation clock past its safe-time. However, real messages with timestamps larger than the safe-time are not sent, but are buffered until it is safe to transmit them. Although state saving is required in this scheme, it has the advantage that rollbacks to prior states are local to the LP receiving an out-of-order message. There is no need for anti-messages to counteract prematurely transmitted real messages. Under this scheme, the only messages in the simulation will be real messages which transmit actual signal change information. If desired, a limit can be placed on how far past the safe-time an LP is allowed to advance. This time window will limit the amount of state saving overhead, but must be chosen large enough to prevent circular waiting in feedback loops among the LPs. Numerous alternative synchronization protocols are possible as well (e.g., conservative time windows, time warp, lazy cancellation, optimistic time windows, etc. (11)).
- *Expand the VHDL subset supported by VSIM.* Currently, VSIM supports a very limited subset of the standard VHDL language. This has the effect of limiting the number of circuits that can be built and simulated in parallel with a reasonable expenditure of programming resources. Breeden suggests methods for

implementing two key enhancements: resolution functions and wait statements (4:83-84). Other suggested enhancements include complex procedures, bit vectors, and multi-valued logic (MVL) data types.

- *Improve the basic VSIM simulation cycle.* The basic simulation cycle implemented in VSIM suffers from large overheads and poor performance. Current state-of-the-art sequential VHDL simulators available commercially can simulate a circuit many times faster than the sequential version of VSIM. One potential source of improvement is with improved list management. Detailed instrumentation of the simulation cycle may lead to the discovery of other sources of potential improvement.
- *Improve VSIM's postprocessor.* Breeden recommends several options for improving the postprocessor which transforms the sequential Intermetrics code into models compatible with VSIM (4:83).
- *Implement selective output report generation.* Currently, the only option for generating simulation output in VSIM is to report all signal changes in the simulation one at a time. For large circuits, verifying the correctness of such output files is infeasible. As a minimum, an ability should be provided to allow selection of which signals to include in the output report. Preferably, formatted output files such as those produced by Intermetrics should be added.

Appendix A. *Acronyms and Definitions*

A.1 Glossary of Acronyms

AFIT	-	Air Force Institute of Technology
ARPA	-	Advanced Research Projects Agency
IVAN	-	Intermediate VHDL Attributed Notation
LP	-	Logical Process
MFA	-	Mean Field Annealing
PDES	-	Parallel Discrete Event Simulation
SA	-	Simulated Annealing
SBF	-	Simple Breadth-First partitioning
SDF	-	Simple Depth-First partitioning
SDP	-	Simple Data Partitioning
SGE	-	Synopsis Graphic Editor
SPECTRUM	-	Simulation Protocol Evaluation on a Concurrent Testbed using ReUsable Modules
TIG	-	Task Interaction Graph
TPG	-	Task Precedence Graph
VHDL	-	VHSIC Hardware Description Language
VHSIC	-	Very High Speed Integrated Circuit
VLSI	-	Very Large Scale Integrated

A.2 Definitions

Activity - The state of an entity over an interval of time (18:135). For example, the activity of a signal is defined as the sequence of state changes for that signal over a given time period.

Behavior - In VHDL, a *behavior* is an executable process representing a logic gate, input signal, output signal, or other simple VHDL process.

Design Hierarchy - In VHDL, the *design hierarchy* represents the successive decomposition of a design entity into components, binding those components to other design entities that may be decomposed in a similar manner. Collectively, they represent a complete design and are referred to as a design hierarchy (8:2-11).

Event - An activity that causes a change in the state of the simulation model (11). In the context of this thesis, a simulation event is defined as the changing of a signal value from one state to another.

Message - A message is the mechanism used by processes to communicate the modified state information caused by a simulation event. In this thesis, the term *message* implies communications between *logical processes*, and thus, corresponds to inter-processor communications.

Model - An abstract representation of a physical system (1). A model consists of entities and their inter-relationships (18:135).

Process - A succession of entity states over a contiguous time period (18:136). A logical process (LP) is the model's representation of a physical process (PP) in the system (7:198-199).

Signal - In VHDL, a signal represents an object that holds a value and corresponds directly to a metal interconnection within a circuit (8:2-12). Signals define the pathways among VHDL processes (i.e. behaviors) (15:9).

State - The sum of all variables describing an entity at a given instant in time (18:135).

System - A real-world process being modeled and simulated (1).

Appendix B. *AFIT Parallel VHDL Simulation User's Guide*

B.1 Overview.

To execute a parallel VHDL simulation, the VHDL circuit is first compiled with the Intermetrics VHDL simulator. The intermediate C code is then intercepted and transformed into models compatible with AFIT's parallel VHDL simulator - VSIM. VSIM has a sequential mode that can be executed on a single processor system, and a parallel mode that runs on the Intel iPSC/2 and iPSC/i860 Hypercubes (4). This appendix discusses the general process for successfully executing a parallel VHDL simulation using VSIM and then illustrates this process through a step-by-step example.

B.1.1 Required Files. Figure 57 shows the location of the baselined versions of all VSIM related files other than the VHDL circuit specific files. The files identified in bold are required to compile and execute the parallel version of VSIM. The files attached by dashed lines are executable utility routines used in the code transformation process. These routines are actually run on the SPARC, but are archived on the iPSC/2 along with the rest of the VSIM related files.

With the exception of the file `application.h`, all of the source files listed in Figure 57 can be compiled out of the archive directories. The file `application.h` must be modified to identify the desired number of LPs in the simulation, and should be copied to and compiled from the user's local directory.

A brief description of each source file is listed below (4:91-92).

- `vsim.h` - Header file for `vinit.c`, `vsim.c`, `vtools.c`, and `vspec.c`; modeled after Intermetrics' `simutl.h`.
- `vsim.c` - VSIM main simulation loop and associated functions.

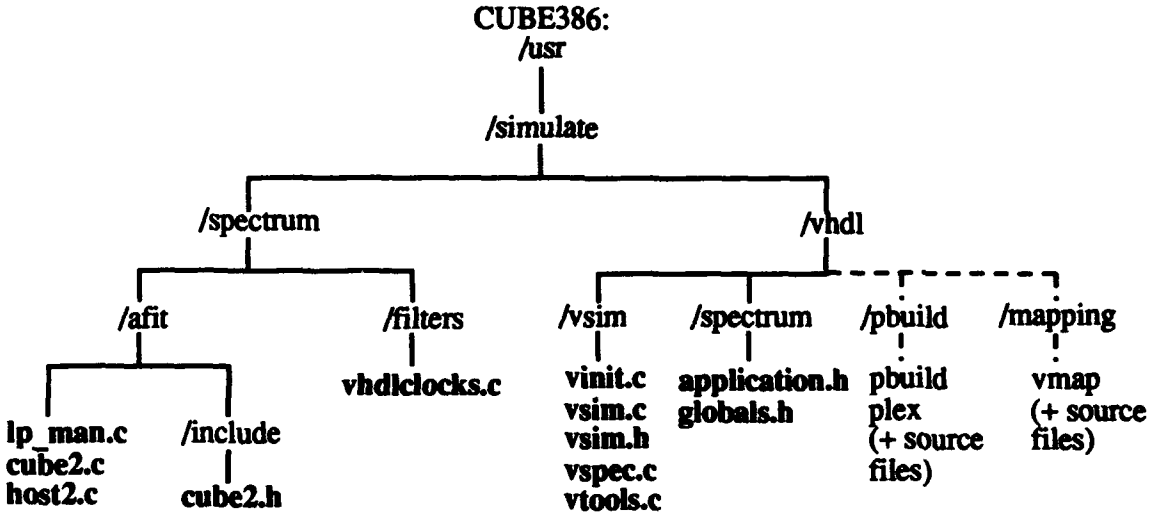


Figure 57. Location of Archived VSIM files on AFIT's iPSC/2 Hypercube

- `vinit.c` - VSIM initialization routines.
- `vtools.c` - Functions for printing VSIM state variables and queues to assist in code maintenance (compilation is optional).
- `vspec.c` - Functions that provide the interface between VSIM and SPECTRUM.
- `globals.h` - Standard SPECTRUM header file. Modified to redefine the event structure.
- `application.h` - Contains application-specific global information for SPECTRUM and `vspec.c`, and is included by `globals.h`. Specifies the number of LPs for a particular simulation run.
- `lp_man.c` - Contains SPECTRUM's LP-level functions.
- `cube2.c` - Provides interface between lp_man.c and the operating system.
- `cube2.h` - Header file for `cube2.c` and `host2.c`.
- `host2.c` - Host program which loads the nodes and starts the simulation.

B.1.2 Process. There are seven basic steps involved in the running of a parallel VHDL simulation with VSIM (4:90):

1. Develop the original VHDL source code describing the circuit to be simulated and the testbench to be used to verify the circuit design. The VHDL source code must comply with the subset of VHDL supported by VSIM as described by Breeden (4).
2. Perform the *Compile*, *Model Generate*, and *Build* phases of the Intermetrics sequential simulation.
3. Using VSIM's postprocessor, `pbuild`, transform the Intermetrics generated source files into VSIM compatible source code.
4. Compile and run the sequential version of VSIM in order to define the behavior dependency relationships.
5. Using the VSIM utility `vmap`, extract the behavior dependency relationships from the output of the sequential VSIM simulation and generate a `.vmap` output file.
6. Using the VSIM Graph-Partitioning Tool (`gp-tool`), read in the `.vmap` file generated in the previous step and generate a logical process dependency file (`lpx.arcs`) and a behavior-to-LP mapping file (`lpx.map`).
7. Compile and run the parallel version of VSIM on the Hypercube.

The remainder of this appendix discusses these seven steps in more detail and concludes with a step-by-step example.

B.2 Generating the VHDL Source Files.

B.2.1 Generating the VHDL Source Code. Step one in the VSIM simulation process is to create the VHDL source code describing the circuit to be simulated. VSIM can only support structural circuit descriptions and simple VHDL processes. Specific limitations are (4:93):

- Signals can only be of type `bit` or `bit-vector`.
- `Bit-vector` signal inputs must be described one bit at a time (e.g., `A(1) <= '0' after 3 ns`).
- In general, processes must be one line descriptions (e.g., `OUT <= IN1 XOR IN2 after delay`). However, a multi-line process (delimited by `begin` and `end process`) may be used if it: 1) waits on all signals, or 2) terminates after the first use.
- Support for VHDL functions and procedures has not been implemented in VSIM. This means that multi-valued logic (MVL) signals and bus resolution functions are not supported. Reference (4:93) for more information.

B.2.2 Establishing an Intermetrics User Library. Step two in the VSIM simulation process involves use of the Intermetrics commercial simulator to compile the VHDL source code and create the sequential simulation models. The Intermetrics compiler is located on `vulcan` in the parallel simulation laboratory. Before establishing a personal library, the environment variables shown in Figure 58 must be included in the user's `.cshrc` file.

The next step is to create the user's individual work library using the sequence of commands shown in Figure 59. These commands must be executed on `vulcan`, substituting the user's own id for "kkapp" (user entries in bold).

The user's work library will only need to be created one time. Once created for the first circuit, this step may be skipped for future circuit simulations.

B.2.3 Compiling, Model Generating, and Building. After a user library has been created, the next steps are to compile the VHDL source code creating an IVAN file as output. The intermediate C source code required by VSIM is ontained by running the Intermetrics model generate routine on the IVAN file. Finally, running the Intermetrics build routine on the intermediate source files creates the required compilation script. The

---

```
#the following lines are for intermetrics vhdl
setenv VHDL_TREE /usr6/vhdl_restore/inter_vhdl/v2.1
setenv VHDL_COMMON /usr6/vhdl_restore/inter_vhdl/v2.1/common
setenv VHDL_HELP_FILE /usr6/vhdl_restore/inter_vhdl/v2.1/common/help.txt
setenv VHDL_LIBROOT /usr6/vhdl_restore/inter_vhdl/v2.1/shiplib
setenv VHDL_LIBSIM /usr6/vhdl_restore/inter_vhdl/v2.1/src/simcore/libsim.a
setenv VHDL_BIN /usr6/vhdl_restore/inter_vhdl/v2.1/bin
set path = ($path /usr6/vhdl_restore/inter_vhdl/v2.1/bin)
```

Figure 58. User `.cshrc` Setup for Running Intermetrics

```
vulcan:~> vls
Standard VHDL 1076 Support Environment Version 2.1 - 1 September 1990
Copyright (C) 1990 Intermetrics, Inc.  All rights reserved.
VLS>makelib -dir=/usr6/vhdl_restore/inter_vhdl/v2.1/shiplib/kkapp <<kkapp>>
VHDVLS-I-CREATED_LIB - Library <<KKAPP>> successfully created.
VLS>define work <<kkapp>>
VLS>setlib <<kkapp>>
VHDVLS-I-DEFAULT_LIBRARY - Default Library is <<KKAPP>>.
VLS>dir
VHDVLS-I-NO_UNITS - No units found in <<KKAPP>>.
VLS>exit
vulcan:~>
```

Figure 59. Setting up User Work Library for Intermetrics

specific steps are listed below.

- Each VHDL source code file must be individually compiled with the Intermetrics `vhdl` command (e.g., `vhdl or_gate.vhd`).
- Each entity/architecture pair must be "model generated" using the Intermetrics `mg` command with the `-debug=cknd` debug switch (e.g., `mg -debug=cknd or_gate(simple)`).
- The top-level configuration is built using the Intermetrics `build` command (e.g., `build '-debug=cknd -replace -ker=assoc_mem assoc_mem_config'`).

B.2.4 Code Transformation. The third step in the VSIM simulation process involves the transformation of the intermediate C source code created by Intermetrics into models that are compatible with VSIM. The inputs for this phase are the intermediate .c and .h files created during the model generate phase, and the compilation script created during the build phase. The compilation script enables the VSIM postprocessor to determine the required files and their correct order of compilation.

VSIM's postprocessor, `pbuild`, is invoked as follows:

```
pbuild script circuit.c
```

where `script` is the name of the compilation script created during the build phase (also referred to as the "`Kernel com`" file), and `circuit` is the name of the circuit being simulated.

The postprocessor, `pbuild`, works by concatenating each of the intermediate .c files into a single file named "`big_circuit.c`" and calling VSIM's lexical analyzer, `plex`, to perform the transformation process, creating the output file `circuit.c`.

The final step in the code transformation is to copy all of the required header files to the same directory as the `circuit.c` file. These header files will also have to be transferred to the target parallel machine with the `circuit.c` file before executing the parallel simulation. The header files can be found in the user's work directory[18].

[18] Unless the user has compiled them into another directory using commands in the VHDL source code.

B.2.5 Transforming Large Circuit Files. A known problem with VSIM's postprocessor is that the `circuit.c` file resulting from the transformation of large circuit simulations may be too large to compile on the Intel Hypercubes. There are two general methods for getting around this problem (4:95):

- Run `plex` on each individual Intermediate C source code file created during the model generate phase. Compile the resulting output into separate object files and link them together to execute VSIM on the hypercubes.

- Construct the original VHDL circuit description using hierarchical configurations. This results in a significant reduction in the size of the corresponding `circuit.c` file. However, the postprocessor is currently unable to properly process the #include directives necessary for the compilation of the `circuit.c` file. These must be manually inserted into the `circuit.c` file, and can be found by an examination of the "`big_circuit.c`" file created by the postprocessor. Both the wallace-tree and the associative-memory were created in this manner.

Refer to (4:95-97) for more detailed information.

B.3 Running Sequential VSIM.

Both the Intermetrics simulator and the VSIM simulator assign each behavior a number at run time. Therefore, before mapping the behaviors onto the parallel architecture, VSIM must be run in sequential mode in order to determine the numbering of the behaviors and the behavior inter-dependency relationships. This is accomplished by using the following compile options in the makefile:

```
-DSPARC -DMAPPING -DOUTPUT
```

The `SPARC` option specifies that the simulation is to be sequential. Note that in the sequential version of VSIM, the SPECTRUM files (`lp_man.c, cube2.c, cube2.h, host2.c, and vhdlclocks.c`) are not required.

The `MAPPING` option specifies that the behavior dependency relationships are to be reported to an output file. When defined, the `MAPPING` option is automatically turned off after the simulation time advances past 0.

The `OUTPUT` option specifies that signal changes are also to be reported to the output file. Output is not required. However, when running a circuit for the first time, it is useful to have output reported for comparison with the Intermetrics output. Note that VSIM does not have a method for selectively deciding which signals to include in the output file - it is an all or nothing option.

Theoretically, if `OUTPUT` is not defined, the sequential simulation can be terminated after simulation time 0 when all of the required dependency information has been reported to the output file. However, there is currently no means implemented for accomplishing this.

B.4 Running Parallel VSIM.

B.4.1 Generating the Partition. Before running the simulation in parallel, the circuit behaviors must be divided into logical processes (LPs), each of which will be assigned to a different processor. Using the output of the sequential simulation which was created with MAPPING defined, the VSIM utility program `vmap` is used to build a file defining the interdependency relationships of the behaviors in the circuit. For example, given the sequential output file `circuit.out`, `vmap` is invoked as follows:

```
vmap circuit.out circuit.vmap
```

where `circuit.vmap` is the `vmap` output file with the following format:

```
behav_id  behav_name  behav_delay  [optional list of dependencies]
```

An example `vmap` output file is shown in Figure 60. If the OUTPUT option was defined, the script `sgrep` can be used to sort the output by time and signal name as follows:

```
sgrep circuit.out output
```

The sorted output data will be in file `output`, and can be compared to the Intermetrics simulation output in order to verify correctness.

The `vmap` output file is used as input to the VHDL Graph-Partitioning Tool (GP-Tool) which builds a directed graph from the behavior dependency information in the file. GP-Tool is a menu-driven utility program which allows the user to select from a variety of partitioning algorithms. Reference the *GP-Tool User's Guide* for detailed instructions on using this utility.

Of the numerous output files created by GP-Tool, the most important are the two files that are required for the parallel execution of VSIM. These are the logical process dependency file (`lpx.arcs`) and a behavior-to-LP mapping file (`lpx.map`). The user will be prompted to enter these file names, and should enter them exactly as shown here, with the "x" replaced by a numeric value specifying the number of LPs in the partition (e.g., `lp8.arcs` and `lp8.map`).

The specification for the `lpx.arcs` file is shown in Figure 61. The `lpx.map` file is a text file containing two columns of numbers. The first column lists each behavior id number, while the second column lists the corresponding LP number to which the behavior is assigned. Examples of an `lpx.arcs` file and an `lpx.map` file are shown in Figures 62 and 63 respectively. Both of these files are read in at run time, and must be in the same directory as the VSIM application.

```
9 ET_DFF_TEST_BENCH(STRUCTURAL) 0 1 2
8 ET_DFF_TEST_BENCH(STRUCTURAL) 0 3
7 ET_DFF(STRUCTURAL) 0
6 ET_DFF(STRUCTURAL) 0
5 NAND_GATE(SIMPLE) 3000000 4 7
4 NAND_GATE(SIMPLE) 3000000 5 6
3 NAND_GATE(SIMPLE) 3000000 0 2
2 THREE_INPUT_NAND_GATE(SIMPLE) 3000000 3 5
1 NAND_GATE(SIMPLE) 3000000 0 2 4
0 NAND_GATE(SIMPLE) 3000000 1
```

Figure 60. Example VMAP Output File

```
0                   # LP index
2                   # Number of input LPs
1 2                 # LP indices of input LPs
0 0                 # Polling frequencies of input LPs
0 0                 # Offset of polling frequency
2                   # Number of input lines
1 2                 # LP number for each input line
2                   # Number of output LPs
2 3                 # LP indices of output LPs
2                   # Number of output lines
2 3                 # LP index for each output line
3000000 5000000     # Minimum delays for each output line
```

Figure 61. Format Specification for `lpx.arcs` Files (4:98)

```
0
1
2
0
0
1
2
1
1
1
1
3000000

1
2
0 2
0 0
0 0
2
0 2
1
2
1
2
3000000

2
1
1
0
0
1
1
2
0 1
2
0 1
3000000 3000000
```

Figure 62. Example `lpx.arcs` File with 3 LPs

```
6 0
5 0
4 0
8 1
3 1
7 1
9 2
1 2
0 2
2 2
```

Figure 63. Example `lpx.map` File with 3 LPs

B.4.2 Execute Parallel Simulation on the Hypercube. The final step in the VSIM parallel simulation process is to copy the necessary files to the target platform, compile the simulation, and execute it on the desired number of nodes. The following files are needed on the hypercube:

- The circuit specific C source code (i.e. the "`circuit.c`" file).
- The header files associated with the "`circuit.c`" file; these header files have file names of the form `FN####`, where `####` is a numeric value determined by the Intermetrics toolset.
- The appropriate `lpx.arcs` and `lpx.map` files for the desired circuit partition.
- A `makefile` to compile the appropriate VSIM, SPECTRUM, and circuit specific files.

In addition, the header file `application.h` will have to be modified to define the desired number of LPs. Thus, `application.h` will have to be copied to, and compiled out of, the user's local directory. It is a good idea to also copy the file `globals.h` to the same local directory. If `application.h` is placed in a directory `~/spectrum` in the user's main directory, the VSIM utility `setlps` can be used to set the number of LPs without requiring the user to manually edit `application.h`. It is invoked as follows:

```
setlps #
```

Where # is the number of LPs desired. However, as currently implemented, `setlps` will only work if the number of LPs is ≤ 9.

The simulation is now ready for compilation on the hypercube. Note that the VSIM code will have to be recompiled each time the number of LPs is changed in `application.h`, but the circuit specific code will have to only be compiled once. The `makefile` should handle this automatically. Once compiled for a given number of LPs, however, the simulation may be run with different partitions by replacing the `lpx.arcs` and `lpx.map` files with no need to recompile.

The simulation should be compiled with the following options:

```
-UMAPPING -UOUTPUT -DCOUNTS -UMONITORCUBE
```

The `MAPPING` option is undefined because the behavior inter-dependency relationships are already known. The `OUTPUT` option is also undefined because the simulation output

creates a performance bottleneck. The AFIT research surrounding VSIM has concentrated on computational speedup and has assumed that other research will adequately address the problems caused by large amounts of output data in parallel simulation applications. However, the OUTPUT option can be defined if the user desires to verify the results of the parallel simulation.

When defined, the COUNTS option causes VSIM to report real, null, and total message counts to the LP's "log" in addition to the normal timing information. If the MONITORCUBE option is defined, each LP will periodically report its simulation time to the terminal screen so that the user can verify that the simulation is progressing. Other options available are the DEBUG and REPORT options which will report very large amounts of data to each LP's "log" file.

After the compilation is complete, the simulation can be started by invoking the `host` program. The user will then be prompted for the name of the circuit program to load, the command line parameters (simulation end time in ns), the number of nodes to use, and the number of LPs in the application (one LP per node). If the number of LPs entered does not match the number in `application.h`, the program exits with an error message.

After the simulation is completed, the simulation timing data will be in a series of "log" files - one for each LP (e.g. `log0`, `log1`, `log2`, ...). These can be concatenated together to provide a summary report for the simulation run. If OUTPUT was defined, each LP's output data will be in an `lpx.out` file (where x is the LP number). These files can be concatenated and then sorted with the `sgrep` utility to provide a file that can be compared with the sequential simulation output.

B.5 Step-by-Step Example.

This section illustrates the VSIM parallel simulation process with a step-by-step example using the edge-triggered D flip-flop (`et_dff`) as the example circuit.

B.5.1 Develop VHDL Source Code. The specific rules and limitations for developing the original VHDL source code are discussed in section B.2.1. Refer to (4) for more detailed information. The VHDL source code files for the `et_dff` circuit are archived on the iPSC/2 (cube386) in the directory `~/usr/simulate/vhdl/et_dff`, and all end in a `.vhd` extension.

B.5.2 Compile, Model Generate, and Build. In this example, it is assumed that the user has already set up an Intermetrics work library as described in section B.2.2. In order to compile, model generate, build, and simulate the circuit with Intermetrics VHDL, a ".com" file similar to the following is required:

```
#!/bin/csh -v
# filename => et_dff.com
vhdl ~/vhdl/aox_gates/nand_nor.vhd
vhdl et_dff.vhd
vhdl et_dff_test_bench.vhd
vhdl et_dff_config.vhd
mg '-debug=cknd nand_gate(simple)'
mg '-debug=cknd three_input_nand_gate(simple)'
mg '-debug=cknd et_dff(structural)'
mg '-debug=cknd et_dff_test_bench(structural)'
mg '-debug=cknd -top et_dff_config'
build '-debug=cknd -replace -ker=et_dff et_dff_config'
sim et_dff
rg et_dff et_dff.rcl
```

In this example, the file is named `et_dff.com` and is executed as follows:

```
~/vhdl/et_dff>  et_dff.com > et_dff.out
```

The activities of the signals specified in the "report control language" file `et_dff.rcl` will be reported in the output file `et_dff.rpt`, and can be used to validate the output of VSIM later.

The file `et_dff.out` will contain a record of the Intermetrics compilation, model generate, and build sessions. This file will contain a list of all the header files (of the form FN####) necessary to compile with VSIM, as well as the "Kernel com" file (the Intermetrics compilation script). To extract this information, the following commands may be used:

```
~/vhdl/et_dff>  more et_dff.out | grep 'H file'
~/vhdl/et_dff>  more et_dff.out | grep Kernel
```

The user should make note of the name of the "Kernel com" file for use with the postprocessor `pbuild`. The header files should be copied to the current directory from the user's work library. For example, if the work directory is `/usr/vhdl/shiplib/kkapp`, the following command can be executed for each header file listed in `et_dff.out`:

```
~/vhdl/et_dff>  cp /usr/vhdl/shiplib/kkapp/FN#### .
```

B.5.3 Run Postprocessor to Transform Code. The "Kernel com" file is the compilation script that the postprocessor uses to build the VSIM compatible code from the Intermetrics code. The output report of the postprocessor is always written to a file called `plex.log` in the same directory. The postprocessor `pbuild` is invoked as follows:

```
~/vhdl/et_dff>  pbuild FN#### et_dff.c
```

Note that `pbuild` concatenates all of the relevant intermediate source code files into one large source file (for this example, it is called "`big_et_dff.c`"), and then performs a series of transformations on it using the lexical utility `plex` to produce the VSIM source file (e.g. `et_dff.c`). In the process of the transformation, not all of the necessary `#include` directives are always included in the transformed source file. They can be extracted from the "big" source file by using the grep command, and then manually inserted into the transformed source file. This usually only happens on large circuits (e.g. the wallace tree multiplier). Reference (4) for more detailed information.

B.5.4 Run Sequential VSIM Simulation. The following files are needed on the sparcstation in order to run the sequential VSIM:

```
~/kkapp/vhdl/vsim/vinit.c      ~/kkapp/vhdl/spectrum/globals.h
                    vsim.c                          application.h
                    vsim.h
                    vspec.c
                    vtools.c
```

To compile VSIM, a `makefile` is required. The example `makefile` below compiles VSIM on the SPARC using the command "`make vsim`."

```
# SPARC macros for sequential execution - type "make vsim"

SPARC_SIMPATH=/usr2/eng/kkapp/vhdl/vsim
SPARC_CKTPATH=/usr2/eng/kkapp/vhdl/et_dff
SPARC_SPECPATH=/usr2/eng/kkapp/vhdl/spectrum

SPARC_OBJS=${SPARC_SIMPATH}/vsim.o   \
           ${SPARC_SIMPATH}/vinit.o  \
           ${SPARC_SIMPATH}/vtools.o \
           ${SPARC_CKTPATH}/et_dff.o

SPARC_CFLAGS=-c -w -g -DSPARC -DMAPPING -DOUTPUT

#
# Compiles VSIM for sequential operation on the Sun SparcStations.
#
vsim: ${SPARC_OBJS}
   $(CC) -o et_dff -g ${SPARC_OBJS}

${SPARC_SIMPATH}/vsim.o: ${SPARC_SIMPATH}/vsim.c \
                         ${SPARC_SIMPATH}/vsim.h
   cd ${SPARC_SIMPATH}; \
   $(CC) ${SPARC_CFLAGS} -I${SPARC_SPECPATH} vsim.c

${SPARC_SIMPATH}/vinit.o: ${SPARC_SIMPATH}/vinit.c \
                          ${SPARC_SIMPATH}/vsim.h
   cd ${SPARC_SIMPATH}; \
   $(CC) ${SPARC_CFLAGS} vinit.c

${SPARC_SIMPATH}/vtools.o: ${SPARC_SIMPATH}/vtools.c \
                           ${SPARC_SIMPATH}/vsim.h
   cd ${SPARC_SIMPATH}; \
   $(CC) ${SPARC_CFLAGS} vtools.c

${SPARC_CKTPATH}/et_dff.o: et_dff.c ${SPARC_SIMPATH}/vsim.h
   $(CC) ${SPARC_CFLAGS} -I${SPARC_SIMPATH} et_dff.c
```

Once the `makefile` has been completed, the following commands compile and run the sequential version of VSIM:

```
~/vhdl/et_dff>  make vsim
~/vhdl/et_dff>  et_dff > temp
```

The output in `temp` will be in time order. However, the following command will do a secondary sort by signal name and place the output in the file `et_dff.out`:

```
~/vhdl/et_dff>  sgrep temp et_dff.out
```

The data in `et_dff.out` can now be compared to the Intermetrics output for accuracy verification. However, the only way to do this is manually, since the two output reports will be in different formats.

B.5.5 Extract Behavior Dependencies using VMAP. Since `MAPPING` was defined during compilation, the output in `temp` also has behavioral information (behavior names, id numbers, and dependencies). Using `vmap`, this information can be filtered out of `temp`

and saved. The vmap program attempts to "guess" the delays of each behavior, based on when dependent behaviors are scheduled. The user is given a chance to override these guesses. In most cases, the behaviors which represent gates show correct delays; the other "system" behaviors should be set to a delay of zero. To run vmap, type the following and respond to the prompts as appropriate (output will be written to et_dff.vmap):

```
~/vhdl/et_dff>  vmap temp et_dff.vmap
```

The mapping of behavior numbers to actual behaviors is done automatically by VSIM. Currently, the only way to verify this mapping is to compare the output of either VSIM or vmap to the schematic.

B.5.6 Generate the Circuit Partition for Parallel Execution. Using the output file from the previous step (e.g., et_dff.vmap), use the VHDL Graph-Partitioning Tool (GP-Tool) to generate the partition for the desired number of LPs. Reference the GP-Tool User's Guide for specific instructions on generating the partition. The needed files from this step are the lpx.arcs and the lpx.map files, where x is the number of LPs.

B.5.7 Compile and Execute the Parallel Simulation. Copy the necessary files specified in section B.4.2 over to the hypercube. Before compiling, a makefile to compile the appropriate files is required. The example makefile below compiles VSIM on the iPSC/2 hypercube using the command "make ipsc."

```
# iPSC macros for parallel execution on iPSC/2 - type "make ipsc"

# local paths
MY_SIMPATH=/usr2/eng/kkapp/vsim
MY_CKTPATH=/usr2/eng/kkapp/et_dff
MY_SPECPATH=/usr2/eng/kkapp/spectrum

# afit paths
AFIT_SIMPATH=/usr/simulate/vhdl/vsim
UVA_SPECPATH=/usr/simulate/spectrum/afit
AFIT_SPECPATH=/usr/simulate/spectrum/afit
AFIT_SPECPATH_INC=/usr/simulate/spectrum/afit/include
AFIT_FILTERPATH=/usr/simulate/spectrum/filters

SPECHEADERS=${MY_SPECPATH}/globals.h ${MY_SPECPATH}/application.h

MY_OBJECTS=${MY_SIMPATH}/vsim.o         \
           ${MY_SIMPATH}/vinit.o        \
           ${MY_SIMPATH}/vtools.o       \
           ${MY_SIMPATH}/vspec.o        \
           ${MY_SPECPATH}/lp_man.o      \
           ${MY_SPECPATH}/cube2.o       \
           ${MY_SPECPATH}/vhdlclocks.o \
           ${MY_CKTPATH}/et_dff.o

MY_CFLAGS=-c -w -UMAPPING -UOUTPUT -DCOUNTS -DMONITORCUBE

ipsc: host node
```

```
host: ${MY_SPECPATH}/host2.o
    $(CC) -o host ${MY_SPECPATH}/host2.o -host

node: ${MY_OBJECTS}
    $(CC) -o et_dff ${MY_OBJECTS} -node

# compiles host2.c out of the archive directories
${MY_SPECPATH}/host2.o: ${AFIT_SPECPATH}/host2.c \
                        ${AFIT_SPECPATH_INC}/cube2.h
    cd ${MY_SPECPATH}; \
    $(CC) ${MY_CFLAGS} -I${AFIT_SPECPATH_INC} ${AFIT_SPECPATH}/host2.c

# compiles vsim.c out of the archive directories
${MY_SIMPATH}/vsim.o: ${AFIT_SIMPATH}/vsim.c \
                      ${AFIT_SIMPATH}/vsim.h  \
                      ${MY_SPECPATH}/globals.h  \
                      ${MY_SPECPATH}/application.h
    cd ${MY_SIMPATH}; \
    $(CC) ${MY_CFLAGS} -I${MY_SPECPATH} ${AFIT_SIMPATH}/vsim.c

# compiles vinit.c out of the archive directories
${MY_SIMPATH}/vinit.o: ${AFIT_SIMPATH}/vinit.c \
                       ${AFIT_SIMPATH}/vsim.h  \
                       ${MY_SPECPATH}/globals.h  \
                       ${MY_SPECPATH}/application.h
    cd ${MY_SIMPATH}; \
    $(CC) ${MY_CFLAGS} -I${MY_SPECPATH} ${AFIT_SIMPATH}/vinit.c

# compiles vtools.c out of the archive directories
${MY_SIMPATH}/vtools.o: ${AFIT_SIMPATH}/vtools.c \
                        ${AFIT_SIMPATH}/vsim.h
    cd ${MY_SIMPATH}; \
    $(CC) ${MY_CFLAGS} ${AFIT_SIMPATH}/vtools.c

# compiles vspec.c out of the archive directories
${MY_SIMPATH}/vspec.o: ${AFIT_SIMPATH}/vspec.c \
                       ${AFIT_SIMPATH}/vsim.h   \
                       ${MY_SPECPATH}/globals.h  \
                       ${MY_SPECPATH}/application.h
    cd ${MY_SIMPATH}; \
    $(CC) ${MY_CFLAGS} -I${MY_SPECPATH} ${AFIT_SIMPATH}/vspec.c

# compiles lp_man.c out of the archive directories
${MY_SPECPATH}/lp_man.o: ${UVA_SPECPATH}/lp_man.c \
                         ${SPECHEADERS}
    cd ${MY_SPECPATH}; \
    $(CC) ${MY_CFLAGS} -I${MY_SPECPATH} ${UVA_SPECPATH}/lp_man.c

# compiles cube2.c out of the archive directories
${MY_SPECPATH}/cube2.o: ${AFIT_SPECPATH}/cube2.c \
                        ${SPECHEADERS}          \
                        ${AFIT_SPECPATH_INC}/cube2.h
    cd ${MY_SPECPATH}; \
    $(CC) ${MY_CFLAGS} -I${AFIT_SPECPATH_INC} -I${MY_SPECPATH}
${AFIT_SPECPATH}/cube2.c

# compiles vhdlclocks.c out of the archive directories
${MY_SPECPATH}/vhdlclocks.o: ${AFIT_FILTERPATH}/vhdlclocks.c \
                             ${SPECHEADERS}
    cd ${MY_SPECPATH}; \
    $(CC) ${MY_CFLAGS} -I${MY_SPECPATH} ${AFIT_FILTERPATH}/vhdlclocks.c
```

```
# compiles et_dff.c out of local directory
${MY_CKTPATH}/et_dff.o: et_dff.c ${AFIT_SIMPATH}/vsim.h
   $(CC) ${MY_CFLAGS} -I${AFIT_SIMPATH} et_dff.c
```

Once the `makefile` has been completed, the following command compiles the parallel version of VSIM:

```
c386 #: setlps #    (where # is the number of LPs desired).
c386 #: make ipsc
```

It should be noted that the exact command depends upon the `makefile`. In the VSIM archives, a `makefile` is provided with each circuit that will compile on the SPARC with the command "`make vsim`," on the iPSC/2 with the command "`make ipsc`," or on the i860 with the command "`make i860`." These makefiles can be used as templates for future circuits.

The simulation is started by invoking the host program and typing the appropriate information at the prompts, including entering the simulation end time in ns as a command line argument. Below is an example simulation session for two LPs (user entries in bold):

```
CUBE386: /usr2/eng/kkapp/et_dff > host
Which application do you want to use?:et_dff
Enter the command line arguments for the program (RETURN if none):
>2000
Is assignment of logical processes to nodes to be from a file? (y/n) -> n

The cube is being used as follows:
CUBENAME        USER     SRM            HOST           TYPE              TTYS
iocube          root     cube386        cube386        0
How many cube nodes do you want to use? (0 to ABORT):2
How many LP's are in this application?:2
Do you want to use the 'natural' node assignment? (y/n): y
Getting cube of size 2 - stand by.
load -H -p 0 0 et_dff 2000
load -H -p 0 1 et_dff 2000
startcube
Cube Loaded
LAST_TIME message from LP 0 on node 0, pid 0.
LAST_TIME message from LP 1 on node 1, pid 0.

End stats messages:
LP 0 (node 0, pid 0):   661 received,   672 sent.
Max message count set at 10, Max messages removed was 2.
LP 1 (node 1, pid 0):   672 received,   661 sent.
Max message count set at 10, Max messages removed was 1.
HOST: Total CPU time waiting: 0.000000 (msecs)
HOST: Wall clock time loading cube: 5 (secs)
HOST: Wall clock time waiting: 2 (secs)
```

Each LP will record its timing data in a "log" file (e.g. `log0` for LP0). With COUNTS defined during compilation, these files will also contain message traffic information. The files can be concatenated and viewed as follows:

```
CUBE386: /usr2/eng/kkapp/et_dff > cat log0 log1 > time2.out
CUBE386: /usr2/eng/kkapp/et_dff > more time2.out

VSIM LP0 reports total time of 863
LP0 NULLs Sent = 652
LP0 NULLs Posted = 0
LP0 NULLs Processed/Deleted From Myself = 0
LP0 NULLs Processed/Deleted From Another LP = 653
LP0 NULLs Annihilated = 0
LP0 Reals Sent = 20
LP0 Reals Posted = 0
LP0 Reals Processed From Myself = 0
LP0 Reals Processed From Another LP = 8
LP 0 wall time taken is 2.107 (secs)
LP 0 messages received 661
LP 0 messages sent 672

VSIM LP1 reports total time of 851
LP1 NULLs Sent = 653
LP1 NULLs Posted = 0
LP1 NULLs Processed/Deleted From Myself = 0
LP1 NULLs Processed/Deleted From Another LP = 652
LP1 NULLs Annihilated = 0
LP1 Reals Sent = 8
LP1 Reals Posted = 0
LP1 Reals Processed From Myself = 0
LP1 Reals Processed From Another LP = 20
LP 1 wall time taken is 2.098 (secs)
LP 1 messages received 672
LP 1 messages sent 661
```

If OUTPUT was defined during compilation, the signal change information for each LP will be in a file called `lpx.out` (where x is the LP number). These files can be concatenated and sorted with `sgrep` for comparison with the sequential output (`et_dff.out`).

Appendix C. *Graph Partitioning Tool (GP-Tool)*

C.1 GP-Tool User's Guide

C.1.1 Overview. Prior to executing a parallel VHDL simulation using VSIM, it is necessary to divide the simulation workload among the available processors. This process, referred to as circuit partitioning, is accomplished by the VHDL Graph-Partitioning Tool (GP-Tool). GP-Tool builds a behavior inter-dependency graph from a circuit description file and provides several partitioning options for assigning the vertices of the inter-dependency graph to the specified number of logical processes (LPs). This section describes how to use GP-Tool to read in a circuit description file and generate the partition output files required by VSIM's parallel mode.

The current version of GP-Tool (version 2.0) is an extension to the *VHDL Graph Searching Program*, henceforth referred to as the original version of GP-Tool. It was written in 1992 by Maj Eric R. Christensen, USA, instructor at the Air Force Institute of Technology, in order to provide a random mapping of the VHDL behaviors onto the logical processes of the parallel simulation. The ability to perform a topological sort on the nodes in the problem-graph was also provided (25). GP-Tool is implemented in the Ada programming language using the Sun Ada Compiler, version 1.1 (available on `aurora` in the AFIT parallel simulation laboratory).

C.1.2 Building the Behavior Inter-Dependency Graph. The introductory screen to GP-Tool is shown in Figure 64. It describes the required format of the circuit description input file and prompts for the input filename (e.g. "et_dff.vmap" in Figure 64). The circuit description file contains a list of circuit behaviors along with their names, logical

```
**************************************************************************

Welcome to the VHDL Graph Partitioning Tool (GP-Tool) - Version 2.0

**************************************************************************

This program reads a file with the following format:
  - Integer, space, String(1..80 characters), Integer, newline
    or 1..N integers followed immediately by a newline
  - The 1..N integers are considered adjacencies to the first integer

The program then builds a graph of the adjacencies and dependencies



Enter the Name of the input data file:

et_dff.vmap

-- Reading Input File and Inserting Vertices in the Graph
-- Reading Input File and Inserting Arcs in the Graph
```

Figure 64. GP-Tool Introductory Screen

```
9 ET_DFF_TEST_BENCH(STRUCTURAL) 0 1 2
8 ET_DFF_TEST_BENCH(STRUCTURAL) 0 3
7 ET_DFF(STRUCTURAL) 0
6 ET_DFF(STRUCTURAL) 0
5 NAND_GATE(SIMPLE) 3000000 4 7
4 NAND_GATE(SIMPLE) 3000000 5 6
3 NAND_GATE(SIMPLE) 3000000 0 2
2 THREE_INPUT_NAND_GATE(SIMPLE) 3000000 3 5
1 NAND_GATE(SIMPLE) 3000000 0 2 4
0 NAND_GATE(SIMPLE) 3000000 1
```

Figure 65. GP-Tool Input File "et_dff.vmap"

delays, and list of dependencies. With this information, GP-Tool builds a directed graph data structure, with each vertex corresponding to a circuit behavior, and each arc representing a unique behavior-to-behavior dependency.

GP-Tool assumes that the input file contains a line for each behavior, and that each behavior description conforms to the required format specification. The current version of GP-Tool does not contain the error detection and recovery mechanisms necessary to compensate for discrepancies in the input file. An incorrect input file may result in erroneous partition output files, or may cause the program execution to be abandoned.

Correctly formatted input files can be produced by following the instructions in sections B.3 and B.4 of the *AFIT Parallel VHDL Simulation User's Guide* (Appendix B) for using the `vmap` utility program. The output files created by `vmap` conform the GP-Tool input file specifications. An example `vmap` output file for the edge-triggered D flip-flop of Figure 18 is shown in Figure 65.

C.1.3 Main Menu Options.

C.1.3.1 Generate Delay and Adjacency Information File. In the original version of GP-Tool, the `lpx.arcs` files were not created directly. Rather, an intermediate

```
********************** GP-TOOL MAIN MENU ************************

Select one of the following operations:
1 : Generate Delay and Adjacency Information File
2 : Generate SGE Data File
3 : Generate Topological Sort File
4 : Generate Strong Components File
5 : Generate Behavior to Logical Process (LP) Mapping File(s)
0 : Quit GP-Tool


Enter your menu choice now:
```

Figure 66. GP-Tool Main Menu

```
*SIZE
 10
*
*SOURCE
8 0
9 0
*
*SERVER
0 3000000
1 3000000
2 3000000
3 3000000
4 3000000
5 3000000
*
*SINK
6 0
7 0
*
*NETWORK
0 1
1 0 2 4
2 3 5
3 0 2
4 5 6
5 4 7
6
7
8 3
9 1 2
*
```

Figure 67. Example Delay and Adjacency File for Edge-Triggered D Flip-Flop

file was created which, along with the `lpx.map` file, was used as input to a separate utility application called `build_arc` which produced the required `lpx.arcs` file. The delay and adjacency information file created by this menu option represents that intermediate description file. An example for the edge-triggered D flip-flop is shown in Figure 67.

However, the application `build_arc` is no longer supported and is unable to handle large input files. As a result, the functionality to produce the `lpx.arcs` files was built into the current version of GP-Tool, obviating the need for this output file. Nevertheless, the option has been retained in the event that it is needed in the future.

C.1.3.2 Generate SGE Data File. The second main menu option allows the user to create a graph description data file that can be read by the commercial Synopsys design analyzer to produce a graphical representation of the input graph that can be displayed using the Synopsys Graphic Editor (SGE). Again, this is a feature of the original version of GP-Tool that was retained for possible future applications. The process for setting up the Synopsys design analyzer and processing the SGE data file produced by GP-Tool to attain the graphic representation is rather complex and is not

```
The Number of Arcs is   15
The Following Nodes have no Inputs
 I8  I9
The Following Nodes have no Outputs
 O6  O7
ADJ  N0  A0
DEP  N0  A1  A3
ADJ  N1  A1  A1  A1
DEP  N1  A0  A9
ADJ  N2  A2  A2
DEP  N2  A1  A3  A9
ADJ  N3  A3  A3
DEP  N3  A2  A8
ADJ  N4  A4  A4
DEP  N4  A1  A5
ADJ  N5  A5  A5
DEP  N5  A2  A4
ADJ  N6  O6
DEP  N6  A4
ADJ  N7  O7
DEP  N7  A5
ADJ  N8  A8
DEP  N8 I8
ADJ  N9  A9  A9
DEP  N9 I9
```

Figure 68. Example SGE Data File for Edge-Triggered D Flip-Flop

discussed here. An example SGE data file produced by GP-Tool for the edge-triggered D flip-flop is shown in Figure 68.

C.1.3.3 Generate Topological Sort File. The third main menu option allows the user to create a topological ordering of the nodes in the behavior inter-dependency graph. This ordering specifies the order in which the circuit behaviors would have to be executed if simulated sequentially. This is another feature of the original version of GP-Tool that was retained for possible future applications. An example topological sort output file for the edge-triggered D flip-flop is shown in Figure 69.

C.1.3.4 Generate Strong Components File. The fourth option on the main menu is for performing a strong component search on the behavior inter-dependency graph. An example output file for the edge-triggered D flip-flop is shown in Figure 70.

```
N9  N8  N2  N1  N3  N7  N6  N5
N4  N0
```

Figure 69. Example Topological Sort File for Edge-Triggered D Flip-Flop

```
GRAPH INFORMATION              - et_dff.vmap
-------------------------------------------------------
The number of vertices in this graph is    : 10
The number of arcs in this graph is        : 15

PARTITION INFORMATION          - Strong Component Search
-------------------------------------------------------
Number of components : 6
Inter-component arcs : 7

The Strong Component sizes are :
    4     2     1     1     1     1

Component Number 0     - Size: 4     - Local Arcs: 6
-------------------------------------------------------
   2     3     0     1

Component Number 1     - Size: 2     - Local Arcs: 2
-------------------------------------------------------
   5     4

Component Number 2     - Size: 1     - Local Arcs: 0
-------------------------------------------------------
   6

Component Number 3     - Size: 1     - Local Arcs: 0
-------------------------------------------------------
   7

Component Number 4     - Size: 1     - Local Arcs: 0
-------------------------------------------------------
   8

Component Number 5     - Size: 1     - Local Arcs: 0
-------------------------------------------------------
   9
```

Figure 70. Example Strong Component File for Edge-Triggered D Flip-Flop

C.1.3.5 Generate Behavior to LP Mapping Files. The fifth option on the GP-Tool main menu takes the user to a sub-menu with options for generating partition files using one of several partitioning strategies implemented in GP-Tool. The GP-Tool behavior mapping sub-menu is shown in Figure 71, and is discussed in further detail in the next section.

C.1.4 Mapping Menu Options.

C.1.4.1 Generate Partitioning Files. Options 1-6 on the GP-Tool behavior mapping sub-menu allow the user to create the circuit partition files `lpx.map` and `lpx.arcs` required to execute the simulation in parallel using VSIM. In addition, a

```
************ GP-TOOL BEHAVIOR MAPPING MENU *************

Select one of the following operations:
1 : Generate Random Partitioning File
2 : Generate Simple Depth-First Partitioning File
3 : Generate Simple Breadth-First Partitioning File
4 : Generate AB1-Annealing Partitioning File
5 : Generate AB2-Annealing Partitioning File
6 : Generate AB3-Annealing Partitioning File
7 : Turn the .MAP and .ARCS output OFF
8 : Modify the Cost Function Parameters
9 : Return to Main Menu
0 : Quit GP-Tool


Enter your menu choice now:
```

Figure 71. GP-Tool Behavior Mapping Sub-Menu

partition statistics file is created such as the one shown in Figure 27. The following partitioning options are available:

- **Random Partition** - Use a random number function to randomly distribute the behaviors among the specified number of LPs, ignoring the behavior inter-dependency relationships. The user will be prompted to input a random stream number between 1 and 100 that is used as an input to the random number generator. This is the only partitioning option that was available in the original version of GP-Tool.

- **Simple Depth-First (SDF) Partition** - Use a depth-first search algorithm to traverse the behavior inter-dependency graph and determine the LP assignments.

- **Simple Breadth-First (SBF) Partition** - Use a breadth-first search algorithm to traverse the behavior inter-dependency graph and determine the LP assignments.

- **AB1-Annealing Partition** - Use the AB border-annealing algorithm to refine an initial SDF partition.

- **AB2-Annealing Partition** - Use the AB border-annealing algorithm to refine an initial SBF partition.

- **AB3-Annealing Partition** - Use the AB border-annealing algorithm to refine an initial random partition.

Each of the AB Annealing partition options require a set of input parameters to control the border annealing process. The parameters that are specific to the AB Annealing algorithm are presented to the user in a sub-menu after the user has specified the number of LPs and entered the appropriate file names.

```
**************** AB-ANNEALING PARAMETERS ****************

The current parameter values are:
1  : Number of Iterations                - 500
2  : Max Number of Worthless Iterations -  50
3  : Load Imbalance Tolerance           -   5.0 %
4  : Ignore Comm_Dist_Factor            - false
5  : Include Hop Weights in Priorities  - false
6  : Log Annealing Data                 - true
7  : Include Deubg Info in Log          - false
8  : Annealing Log Filename             - annealing_data


Enter the line number of the
parameter to update, or zero (0) to continue:
```

Figure 72. GP-Tool AB Annealing Parameters Sub-Menu

The AB Annealing parameters sub-menu is shown in Figure 72. If the default parameter values are satisfactory, the user can enter '0' to begin the partitioning process. Otherwise, the parameter values can be changed by entering the appropriate line number and entering the new value (if appropriate). The specific parameters are as follows:

- **Number of Iterations** - Defines the maximum number of annealing iterations to perform before terminating the process, with a maximum of 1000. Realistically, the default value of 500 provides more than enough iterations to converge to a solution with the circuits used as test cases in this thesis.

- **Max Number of Worthless Iterations** - Defines the maximum number of consecutive iterations with no net improvement in the communications cost portion of the objective cost function which can be processed before the annealing process is terminated. The counter which tracks worthless iterations is reset to zero each time there is an complete iteration with a net improvement in the subject cost function. This value should be large enough to ensure that the series of worthless iterations indicates an actual solution convergence and not just a temporary anomaly in the annealing process.

- **Load Imbalance Tolerance** - Defines the maximum value of the load delta factor H_b that is acceptable. Moves which cause H_b to be larger than the value of this parameter will not be made, even if they would result in a reduction in the communications cost sub-function. A value of 0.0% for this parameter will automatically be defaulted to the value of H_b in the initial partition. A load imbalance of one behavior can result in the initial partition algorithm if the number of behaviors is not evenly divisible by the number of LPs. However, if the number of behaviors *is* divisible by the number of LPs, the initial partition will have an H_b of 0.0% and the a 0.0% value for this parameter will prevent any moves from occurring, rendering the annealing process useless.

- **Ignore_Comm_Dist_Factor** - Boolean value that allows the factor H_d to be ignored when computing the value of the communications sub-function during the annealing process. When false, it is possible to experience a slight increase in the number of `LP_Output_Lines` as the value of H_d is reduced. However, when true, the annealing algorithm will have a tendency to prevent such an increase to the number of `LP_Output_Lines` (which has a direct impact on the number of null messages sent), as well as lead to a larger reduction in the number of inter-LP arcs. Furthermore, if the number of `LP_Output_Lines` was reduced when this parameter was false, setting it to true may lead to a larger reduction.

- **Include Hop Weights in Priorities** - When calculating the communications costs of a partition, each inter-LP arc is multiplied by a hop-weight corresponding to the number of hops in the corresponding physical communications link. When set to true, this parameter will take this weighting into account when prioritizing and queueing behaviors during the annealing process. It should be noted that the default hop weights are all 1.0 (evenly weighted), thus rendering this option meaningless. The hop weights can be modified using option eight on the behavior mapping sub-menu. If uneven hop weights are used, setting this parameter to true has a tendency to deteriorate the performance of the annealing algorithm, with no noticeable improvement in the solution quality.

- **Log Annealing Data** - Boolean value that controls the printing of the partition statistics values to an output file for the initial partition and after each annealing iteration. This allows the progress of the annealing algorithm to be examined.

- **Include Debug Info in Log** - Boolean value that will cause information to be added to the annealing log file for each behavior that is removed from the annealing queue. This is for development/debugging purposes only. When true, the number of iterations should be reduced to the 1-5 range, or the annealing log will become too large to be of practical value. This parameter will have no effect if the previous parameter is set to false.

- **Annealing Log Filename** - Defines the name of the annealing log output file.

C.1.4.2 Toggle .MAP and .ARCS Output. Option seven on the GP-Tool behavior mapping sub-menu allows the user to toggle the creation of the `lpx.map` and `lpx.arcs` files on and off. This allows the user to create only the partition statistics files for comparison purposes without incurring the overhead of entering filenames and creating the `lpx.map` and `lpx.arcs` files. It is included primarily as an aid to the development and testing process. It should be noted that when the `lpx.arcs` file is not produced, the value of `Larcs` is set to 1.0 in the calculation of the predicted simulation speedup in the partition statistics file because the actual lookahead values are *not* available.

C.1.4.3 Modify Cost Function Parameters. Option eight on the GP-Tool behavior mapping sub-menu allows the user to set several miscellaneous parameters that

```
************* MODIFY PARAMETERS MENU *************

The current parameter values are:
1  : Consider Topological Variation    - false
2  : Ignore Zero Delays in .arcs file  - true
3  : Alpha             - 100.00
4  : Beta              - 1.00
5  : Gamma             - 0.0750000000
6  : One_Hop_Weight    - 1.0
7  : Two_Hop_Weight    - 1.0
8  : Three_Hop_Weight  - 1.0
9  : Four_Hop_Weight   - 1.0
10 : Five_Hop_Weight   - 1.0
11 : Six_Hop_Weight    - 1.0
12 : Seven_Hop_Weight  - 1.0

Enter the line number of the
parameter to update, or zero (0) to exit:
```

Figure 73. GP-Tool Cost Function Parameters Sub-Menu

are not specific to the AB-Annealing algorithm. The modify parameters sub-menu is shown in Figure 73. The specific parameters include the following:

- **Consider Topological Variation** - Boolean value which controls whether or not the topological layout of the hypercube is considered when building an SDF or SBF partition. Reference section 3.4.4 for more information.

- **Ignore Zero Delays in .arcs File** - Boolean value which controls how zero-delay behaviors are handled during the calculation of the LP delay values for the `lpx.arcs` file. If true, a source behavior with a logical delay of zero in LP A with an external arc to LP B will not cause the lookahead value for the LP output line from A to B to be set to zero. Rather, the smallest non-zero value calculated for that output line will be used. This works due to the fact that in VSIM, all source behaviors must have their signal changes explicitly defined in the testbench, thus causing them to be placed in the active list at simulation startup.

- **Alpha, Beta, and Gamma** - Coefficients to the partition cost function that are used in the calculation of the predicted speedup. These values have no effect upon the partitioning algorithms.

- **Hop Weights** - Weights associated with each inter-LP arc based upon the number of hops in the corresponding physical communications link. These hop weights will not affect the random, SDF, or SBF partitions other than increasing the value of the communications cost function. However, since the communications cost function is actively used in the AB border annealing process, the hop weights will directly affect the resulting AB annealing partition.

C.2 GP-Tool Developer's Guide

This section provides a brief description of the Ada source files required to compile and run the current version of the GP-Tool utility. The source code is highly modularized, with different functions performed by separate Ada packages. The following is a list of the primary Ada source files in the GP-Tool hierarchy:

```
ab_annealing_pkg_b.a        printing_pkg_b.a          sdf_partition_pkg_b.a
ab_annealing_pkg_s.a        printing_pkg_s.a          sdf_partition_pkg_s.a
annealing_tools_pkg_b.a     rand_gen_b.a              sort_vhdl_p.a
annealing_tools_pkg_s.a     rand_gen_s.a              statistics_pkg_b.a
build_graph_b.a             random_partition_b.a      statistics_pkg_s.a
build_graph_s.a             random_partition_s.a      tools_pkg_b.a
graph_tool.a                sbf_partition_pkg_b.a     tools_pkg_s.a
misc_vhdl_pkgs_s.a          sbf_partition_pkg_s.a     vhdl_top_sort_p.a
print_graph_b.a             sc_search_pkg_b.a
print_graph_s.a             sc_search_pkg_s.a
```

The remaining files required to compile GP-Tool fall into the category of generic Ada packages, and are as follows:

```
digraph_utilities_b.a          list_search_b.a
digraph_utilities_s.a          list_search_s.a
directed_graph_b.a             list_utilities_b.a
directed_graph_s.a             list_utilities_s.a
gen_doubly_linked_list_b.a     map_unbounded_cache_b.a
gen_doubly_linked_list_s.a     map_unbounded_cache_s.a
gen_static_strings_b.a         priority_queue_b.a
gen_static_strings_s.a         priority_queue_s.a
generic_queue_b.a              seq_storage_mngr_b.a
generic_queue_s.a              seq_storage_mngr_s.a
generic_top_sort_b.a           set_iterator_b.a
generic_top_sort_s.a           set_iterator_s.a
lim_private_map_pkg_b.a        stack_pkg_b.a
lim_private_map_pkg_s.a        stack_pkg_s.a
```

Figure 74 shows how the various Ada packages interact to form the complete GP-Tool application. Note that `misc_vhdl_pkgs_s.a` contains several package instantiations in a single file. Note also that the figure does not specify the specific generic package dependencies, but treats all of the generic packages as a single group in order to simplify the diagram.

The functionality of the most critical packages are listed below. Each of these packages also contains extensive source file comments with more detailed information.

- **`graph_tool`** - provides the overall program flow control, displaying the menus and making the appropriate sub-routine calls based upon the menu option selected by the user.

- **`build_graph`** - reads the input file and builds the behavior inter-dependency graph structure in memory.

- **`random_partition_pkg`** - performs a random mapping of the behaviors to the given number of LPs.

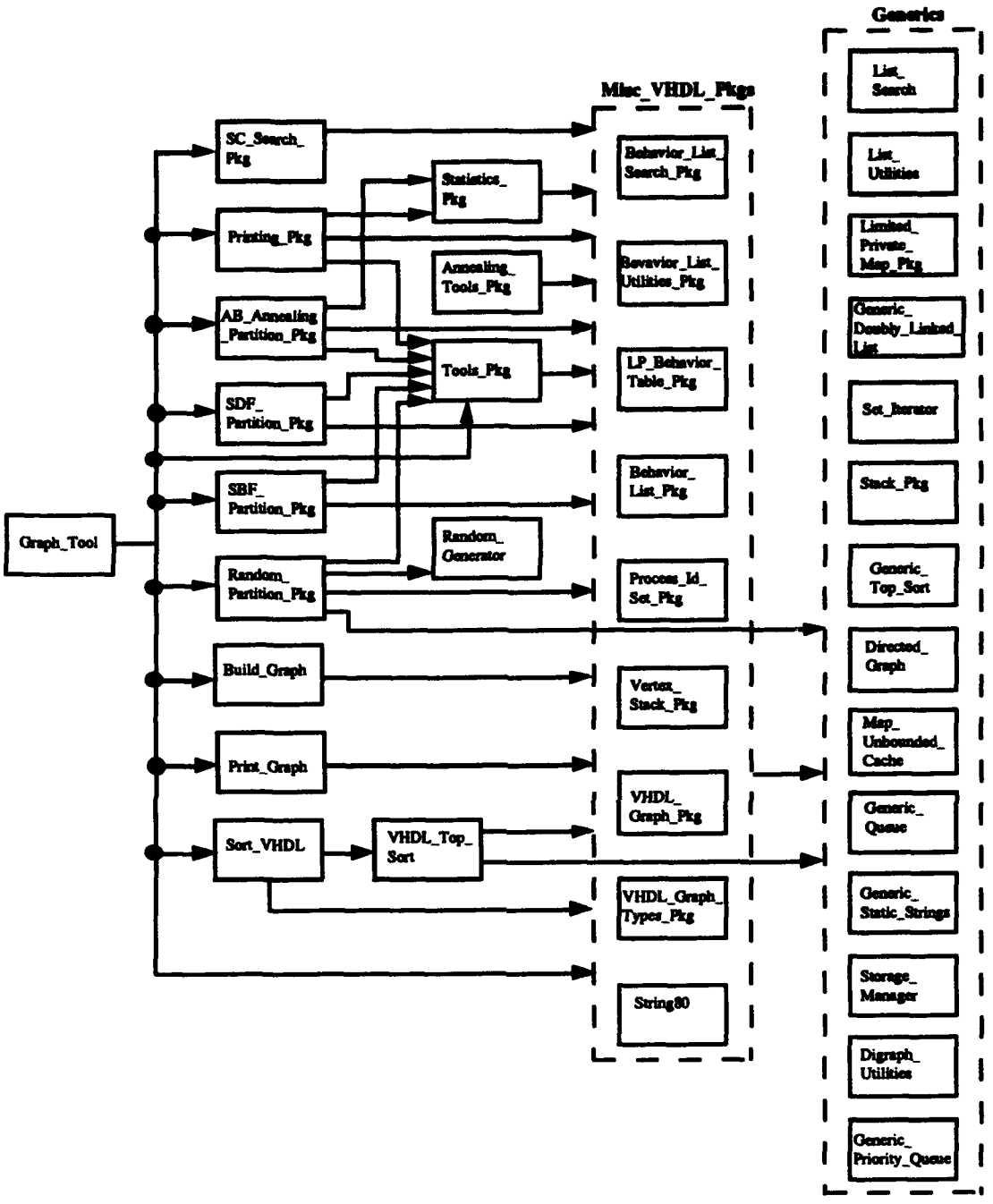


Figure 74. GP-Tool Ada Package Dependency Graph

- **sdf_partition_pkg** - performs a simple depth-first (SDF) mapping of the behaviors to the given number of LPs.
- **sbf_partition_pkg** - performs a simple breadth-first (SBF) mapping of the behaviors to the given number of LPs.

- **ab_annealing_partition_pkg** - takes a partitioned graph as input and performs the AB border annealing algorithm in an effort to improve the quality of the partition.

- **annealing_tools_pkg** - provides several utility functions used by the AB border annealing algorithm, such as displaying the annealing parameters sub-menu, initializing the vertex priorities, and printing relevant data and debugging information to the annealing log.

- **tools_pkg** - consolidates several utility functions used by all of the partitioning algorithms into a single file in order to minimize code duplication and improve code maintainability; functions include initializing data structures and linking a new vertices to the Parent-Child chains representing a given LP assignment.

- **statistics_pkg** - provides routines needed to evaluate a partitioned graph and calculate the statistics values associated with the partition (inter-component arcs, load imbalance, predicted speedup, etc.); includes a routine to build and initialize the communications weight matrix.

- **printing_pkg** - provides routines to print the key output files including the partition statistics file, the `lpx.map` file, and the `lpx.arcs` file.

- **sc_search_pkg** - provides routines to perform a strong component search on an unpartitioned input file, linking the strong components together similar to the LP assignment lists.

- **print_graph** - provides routine to print the "queue.dat" file used as input to the `build_arc` utility which can produce the corresponding `lpx.arcs` file; note that `build_arc` is no longer supported and cannot handle large input graphs.

- **sort_VHDL** - provides routines to perform a topological sort on an input graph.

Appendix D. *Simulation Performance Data*

This appendix contains additional simulation performance data to supplement the data appearing in Chapter 5. The first section contains tables which summarize the actual performance data for a selected subset of test cases. The second section contains additional message traffic analysis graphs for a selected subset of test cases.

Table 3 summarizes the performance of the single LP wallace tree multiplier case. Tables 4 to 16 summarize the performance of the wallace tree multiplier for all four partitioning types: random, SDF, SBF, and AB border annealing. Tables are included for 2, 4, and 8 LPs for each partition type. Each table contains the simulation time, the total time (which includes the time to load the cube nodes), real message transmitted, null messages transmitted, and total messages transmitted. All performance measurements are with respect to the simulation time. All times are in ns. Each table also summarizes the corresponding partition statistics values as calculated by GP-Tool. Tables 17 to 28 provide an identical set of data for the associative memory array.

Figures 75 to 78 provide additional real messages vs. null messages transmitted graphs to supplement the test cases discussed in Chapter 5. Graphs are provided for both the wallace tree multiplier and the associative memory array. Figures 79 to 84 provide the corresponding total messages transmitted vs. output arcs graphs, while Figures 85 to 90 provide the total messages transmitted vs. LP output lines graphs.

Table 3. Wallace Tree 1 LP Simulation Results

Circuit	- Wallace
Partition	- Random
Num_Vertices	- 1,050
Num_Arcs	- 1,770
Num_LPs	- 1
Inter-LP Arcs	- 0
Wght_Inter_LP_Arcs	- 0.0
Avg_Wght_Arcs	- 0.0
Stddev_Wght_Out_Arcs	- 0.0
Maxdev_Wght_Out_Arcs	- 0.0
Stddev_Wght_In_Arcs	- 0.0
Maxdev_Wght_In_Arcs	- 0.0
Comm_Cost_Factor	- 0.00%
Comm_Dist_Factor	- 0.00%
LP_Output_Lines	- 0
Lookahead_Factor	- 0.000
Avg_Comp_Load	- 1050.0
Stddev_Comp_Load	- 0.0
Maxdev_Comp_Load	- 0.0
Load_Delta_Factor	- 0.00%
Predicted Speedup	- 1.00

Trial	Sim Time (ns)	Total Time (ns)	Reals Sent	Nulls Sent	Total Sent
1	67,940	77,044	0	0	0
2	67,940	77,032	0	0	0
3	67,940	77,022	0	0	0
4	67,940	77,036	0	0	0
5	67,941	77,034	0	0	0
6	67,950	77,042	0	0	0
7	67,949	77,036	0	0	0
8	67,949	77,041	0	0	0
9	67,950	77,037	0	0	0
10	67,949	77,032	0	0	0
11	67,950	77,075	0	0	0
12	67,950	77,053	0	0	0
13	67,950	77,049	0	0	0
14	67,950	77,263	0	0	0
15	67,950	77,208	0	0	0
16	67,950	77,051	0	0	0
17	67,949	77,185	0	0	0
18	67,950	78,629	0	0	0
19	67,950	77,517	0	0	0
20	67,948	77,073	0	0	0
Avg	67,947	77,173	0	0	0
Stddev	4	354	0	0	0

Table 4. Wallace Tree 2 LP Random Partition Simulation Results

Circuit	- Wallace
Partition	- Random
Num_Vertices	- 1,050
Num_Arcs	- 1,770
Num_LPs	- 2
Inter-LP Arcs	- 866
Wght_Inter_LP_Arcs	- 866.0
Avg_Wght_Arcs	- 433.0
Stddev_Wght_Out_Arcs	- 5.7
Maxdev_Wght_Out_Arcs	- 4.0
Stddev_Wght_In_Arcs	- 5.7
Maxdev_Wght_In_Arcs	- 4.0
Comm_Cost_Factor	- 48.93%
Comm_Dist_Factor	- 0.92%
LP_Output_Lines	- 2
Lookahead_Factor	- 1.000
Avg_Comp_Load	- 525.0
Stddev_Comp_Load	- 0.0
Maxdev_Comp_Load	- 0.0
Load_Delta_Factor	- 0.00%
Predicted Speedup	- 1.84

Trial	Sim Time (ms)	Total Time (ms)	Reals Sent	Nulls Sent	Total Sent
1	29,609	37,997	8,502	2,777	11,279
2	29,245	37,638	8,502	2,774	11,276
3	29,597	37,979	8,502	2,763	11,265
4	29,954	38,343	8,502	2,783	11,285
5	29,753	38,142	8,502	2,771	11,273
6	29,862	38,277	8,502	2,741	11,243
7	30,096	38,497	8,502	2,778	11,280
8	30,607	39,645	8,502	2,749	11,251
9	29,853	40,482	8,502	2,760	11,262
10	29,523	38,606	8,502	2,813	11,315
11	29,609	38,008	8,502	2,801	11,303
12	30,055	38,449	8,502	2,790	11,292
13	29,702	38,091	8,502	2,764	11,266
14	29,825	38,223	8,502	2,812	11,314
15	29,861	38,248	8,502	2,767	11,269
16	29,850	38,239	8,502	2,771	11,273
17	29,941	38,337	8,502	2,776	11,278
18	29,624	38,015	8,502	2,782	11,284
19	30,130	38,524	8,502	2,737	11,239
20	29,722	38,114	8,502	2,766	11,268
Avg	29,821	38,393	8,502	2,774	11,276
Stddev	275	613	0	20	20

Table 5. Wallace Tree 4 LP Random Partition Simulation Results

Circuit	- Wallace
Partition	- Random
Num_Vertices	- 1,050
Num_Arcs	- 1,770
Num_LPs	- 4
Inter-LP Arcs	- 1,332
Wght_Inter_LP_Arcs	- 1332.0
Avg_Wght_Arcs	- 333.0
Stddev_Wght_Out_Arcs	- 21.1
Maxdev_Wght_Out_Arcs	- 21.0
Stddev_Wght_In_Arcs	- 14.5
Maxdev_Wght_In_Arcs	- 16.0
Comm_Cost_Factor	- 75.25%
Comm_Dist_Factor	- 6.31%
LP_Output_Lines	- 12
Lookahead_Factor	- 1.000
Avg_Comp_Load	- 262.5
Stddev_Comp_Load	- 0.6
Maxdev_Comp_Load	- 0.5
Load_Delta_Factor	- 0.19%
Predicted Speedup	- 2.13

Trial	Sim Time (ms)	Total Time (ms)	Reals Sent	Nulls Sent	Total Sent
1	24,989	34,176	14,902	17,831	32,733
2	26,347	35,510	14,902	17,807	32,709
3	26,585	35,821	14,902	17,777	32,679
4	24,921	34,167	14,902	17,978	32,880
5	25,690	34,863	14,902	17,880	32,782
6	25,949	35,134	14,902	17,739	32,641
7	25,127	34,383	14,902	17,834	32,736
8	25,256	34,425	14,902	17,904	32,806
9	24,970	34,145	14,902	17,813	32,715
10	24,774	34,030	14,902	17,900	32,802
11	26,276	35,487	14,902	17,937	32,839
12	25,069	34,255	14,902	18,020	32,922
13	26,833	36,065	14,902	17,803	32,705
14	26,263	35,505	14,902	17,841	32,743
15	24,916	34,095	14,902	17,794	32,696
16	25,645	34,890	14,902	17,932	32,834
17	26,026	35,276	14,902	17,646	32,548
18	24,894	34,127	14,902	17,895	32,797
19	26,605	35,771	14,902	17,842	32,744
20	24,827	34,072	14,902	17,938	32,840
Avg	25,598	34,810	14,902	17,856	32,758
Stddev	684	682	0	85	85

Table 6. Wallace Tree 8 LP Random Partition Simulation Results

Circuit	- Wallace
Partition	- Random
Num_Vertices	- 1,050
Num_Arcs	- 1,770
Num_LPs	- 8
Inter-LP Arcs	- 1,558
Wght_Inter_LP_Arcs	- 1558.0
Avg_Wght_Arcs	- 194.8
Stddev_Wght_Out_Arcs	- 11.1
Maxdev_Wght_Out_Arcs	- 12.3
Stddev_Wght_In_Arcs	- 10.0
Maxdev_Wght_In_Arcs	- 13.3
Comm_Cost_Factor	- 88.02%
Comm_Dist_Factor	- 6.29%
LP_Output_Lines	- 56
Lookahead_Factor	- 1.000
Avg_Comp_Load	- 131.3
Stddev_Comp_Load	- 0.5
Maxdev_Comp_Load	- 0.8
Load_Delta_Factor	- 0.57%
Predicted Speedup	- 1.39

Trial	Sim Time (ms)	Total Time (ms)	Reals Sent	Nulls Sent	Total Sent
1	55,859	67,535	19,066	81,967	101,033
2	55,618	67,278	19,066	81,712	100,778
3	56,053	67,753	19,066	81,763	100,829
4	55,725	67,393	19,066	82,079	101,145
5	55,769	67,480	19,066	81,736	100,802
6	54,848	66,499	19,066	82,010	101,076
7	56,418	68,087	19,066	81,496	100,562
8	56,055	67,750	19,066	81,585	100,651
9	55,276	66,950	19,066	82,129	101,195
10	54,710	66,359	19,066	81,845	100,911
11	55,025	66,714	19,066	81,707	100,773
12	54,550	66,234	19,066	82,219	101,285
13	53,682	65,293	19,066	82,543	101,609
14	54,338	65,950	19,066	82,155	101,221
15	54,340	65,957	19,066	81,521	100,587
16	54,788	66,491	19,066	82,530	101,596
17	55,174	66,784	19,066	81,883	100,949
18	54,459	66,058	19,066	81,846	100,912
19	56,168	67,836	19,066	83,245	102,311
20	55,859	67,455	19,066	81,846	100,912
Avg	55,236	66,893	19,066	81,991	101,057
Stddev	736	754	0	403	403

Table 7. Wallace Tree 2 LP SDF Partition Simulation Results

Circuit	- Wallace
Partition	- SDF
Num_Vertices	- 1,050
Num_Arcs	- 1,770
Num_LPs	- 2
Inter-LP Arcs	- 150
Wght_Inter_LP_Arcs	- 150.0
Avg_Wght_Arcs	- 75.0
Stddev_Wght_Out_Arcs	- 97.6
Maxdev_Wght_Out_Arcs	- 69.0
Stddev_Wght_In_Arcs	- 97.6
Maxdev_Wght_In_Arcs	- 69.0
Comm_Cost_Factor	- 8.47%
Comm_Dist_Factor	- 92.00%
LP_Output_Lines	- 2
Lookahead_Factor	- 1.000
Avg_Comp_Load	- 525.0
Stddev_Comp_Load	- 0.0
Maxdev_Comp_Load	- 0.0
Load_Delta_Factor	- 0.00%
Predicted Speedup	- 1.94

Trial	Sim Time (ms)	Total Time (ms)	Reals Sent	Nulls Sent	Total Sent
1	29,625	38,299	1,027	1,928	2,955
2	29,357	37,968	1,027	1,902	2,929
3	29,537	38,150	1,027	1,927	2,954
4	29,553	38,170	1,027	1,921	2,948
5	29,595	38,205	1,027	1,891	2,918
6	29,597	38,220	1,027	1,950	2,977
7	29,597	38,229	1,027	1,897	2,924
8	29,621	38,257	1,027	1,894	2,921
9	29,669	38,288	1,027	1,869	2,896
10	29,513	38,143	1,027	1,956	2,983
11	29,385	38,034	1,027	1,923	2,950
12	29,564	38,172	1,027	1,942	2,969
13	29,487	38,102	1,027	1,916	2,943
14	29,537	38,152	1,027	1,917	2,944
15	29,540	38,152	1,027	1,913	2,940
16	29,660	38,306	1,027	1,927	2,954
17	29,694	38,281	1,027	1,914	2,941
18	29,652	38,259	1,027	1,900	2,927
19	29,679	38,284	1,027	1,917	2,944
20	29,443	38,058	1,027	1,947	2,974
Avg	29,565	38,183	1,027	1,917	2,944
Stddev	92	89	0	20	20

Table 8. Wallace Tree 4 LP SDF Partition Simulation Results

Circuit	- Wallace
Partition	- SDF
Num_Vertices	- 1,050
Num_Arcs	- 1,770
Num_LPs	- 4
Inter-LP Arcs	- 312
Wght_Inter_LP_Arcs	- 312.0
Avg_Wght_Arcs	- 78.0
Stddev_Wght_Out_Arcs	- 66.5
Maxdev_Wght_Out_Arcs	- 89.0
Stddev_Wght_In_Arcs	- 46.6
Maxdev_Wght_In_Arcs	- 49.0
Comm_Cost_Factor	- 17.63%
Comm_Dist_Factor	- 114.10%
LP_Output_Lines	- 11
Lookahead_Factor	- 0.667
Avg_Comp_Load	- 262.5
Stddev_Comp_Load	- 0.6
Maxdev_Comp_Load	- 0.5
Load_Delta_Factor	- 0.19%
Predicted Speedup	- 3.01

Trial	Sim Time (ms)	Total Time (ms)	Reals Sent	Nulls Sent	Total Sent
1	25,313	34,688	2,788	14,867	17,655
2	23,748	33,012	2,788	14,921	17,709
3	23,785	33,096	2,788	15,123	17,911
4	23,729	33,070	2,788	15,003	17,791
5	23,794	33,163	2,788	15,111	17,899
6	23,531	32,799	2,788	14,115	16,903
7	24,195	33,405	2,788	14,289	17,077
8	22,935	32,238	2,788	14,075	16,863
9	24,662	33,928	2,788	14,150	16,938
10	22,591	31,860	2,788	14,118	16,906
11	23,626	32,926	2,788	14,241	17,029
12	24,506	33,787	2,788	13,910	16,698
13	22,726	32,008	2,788	14,111	16,899
14	23,479	32,755	2,788	14,216	17,004
15	22,690	31,980	2,788	14,125	16,913
16	22,976	32,279	2,788	14,038	16,826
17	24,536	33,746	2,788	14,109	16,897
18	25,073	34,369	2,788	14,004	16,792
19	23,013	32,217	2,788	14,050	16,838
20	23,050	32,320	2,788	14,009	16,797
Avg	23,698	32,982	2,788	14,329	17,117
Stddev	785	793	0	402	402

Table 9. Wallace Tree 8 LP SDF Partition Simulation Results

Circuit	- Wallace
Partition	- SDF
Num_Vertices	- 1,050
Num_Arcs	- 1,770
Num_LPs	- 8
Inter-LP Arcs	- 536
Wght_Inter_LP_Arcs	- 536.0
Avg_Wght_Arcs	- 67.0
Stddev_Wght_Out_Arcs	- 49.6
Maxdev_Wght_Out_Arcs	- 121.0
Stddev_Wght_In_Arcs	- 18.5
Maxdev_Wght_In_Arcs	- 40.0
Comm_Cost_Factor	- 30.28%
Comm_Dist_Factor	- 180.60%
LP_Output_Lines	- 45
Lookahead_Factor	- 0.533
Avg_Comp_Load	- 131.3
Stddev_Comp_Load	- 0.5
Maxdev_Comp_Load	- 0.8
Load_Delta_Factor	- 0.57%
Predicted Speedup	- 2.17

Trial	Sim Time (ms)	Total Time (ms)	Reals Sent	Nulls Sent	Total Sent
1	44,284	55,837	5,241	60,315	65,556
2	43,963	55,140	5,241	60,075	65,316
3	45,067	56,310	5,241	60,047	65,288
4	46,853	57,940	5,241	60,370	65,611
5	44,776	55,888	5,241	60,344	65,585
6	44,770	56,316	5,241	60,019	65,260
7	44,765	56,304	5,241	58,631	63,872
8	45,025	56,602	5,241	58,900	64,141
9	46,378	58,034	5,241	59,906	65,147
10	45,725	57,298	5,241	59,544	64,785
11	46,841	58,495	5,241	58,854	64,095
12	44,336	55,741	5,241	59,616	64,857
13	46,816	58,428	5,241	59,185	64,426
14	44,544	56,215	5,241	59,027	64,268
15	45,929	57,585	5,241	59,265	64,506
16	44,008	55,577	5,241	59,257	64,498
17	43,400	55,065	5,241	59,553	64,794
18	44,993	56,582	5,241	59,666	64,907
19	47,285	58,825	5,241	58,797	64,038
20	47,122	58,614	5,241	59,688	64,929
Avg	45,344	56,840	5,241	59,553	64,794
Stddev	1,156	1,180	0	533	533

Table 10. Wallace Tree 2 LP SBF Partition Simulation Results

Circuit	- Wallace
Partition	- SBF
Num_Vertices	- 1,050
Num_Arcs	- 1,770
Num_LPs	- 2
Inter-LP Arcs	- 162
Wght_Inter_LP_Arcs	- 162.0
Avg_Wght_Arcs	- 81.0
Stddev_Wght_Out_Arcs	- 97.6
Maxdev_Wght_Out_Arcs	- 69.0
Stddev_Wght_In_Arcs	- 97.6
Maxdev_Wght_In_Arcs	- 69.0
Comm_Cost_Factor	- 9.15%
Comm_Dist_Factor	- 85.19%
LP_Output_Lines	- 2
Lookahead_Factor	- 1.000
Avg_Comp_Load	- 525.0
Stddev_Comp_Load	- 0.0
Maxdev_Comp_Load	- 0.0
Load_Delta_Factor	- 0.00%
Predicted Speedup	- 1.94

Trial	Sim Time (ms)	Total Time (ms)	Reals Sent	Nulls Sent	Total Sent
1	30,379	38,807	1,088	1,955	3,043
2	30,352	38,749	1,088	1,998	3,086
3	30,176	38,582	1,088	2,008	3,096
4	30,135	38,532	1,088	1,990	3,078
5	30,173	38,645	1,088	1,988	3,076
6	29,971	38,391	1,088	2,016	3,104
7	30,229	38,777	1,088	1,964	3,052
8	30,121	38,670	1,088	2,011	3,099
9	30,405	39,714	1,088	1,954	3,042
10	30,003	38,617	1,088	1,992	3,080
11	30,173	38,588	1,088	1,963	3,051
12	30,357	38,774	1,088	2,003	3,091
13	30,233	39,110	1,088	1,983	3,071
14	29,955	38,449	1,088	1,987	3,075
15	30,023	38,731	1,088	2,016	3,104
16	30,107	38,587	1,088	2,012	3,100
17	30,141	38,820	1,088	1,940	3,028
18	30,189	38,970	1,088	1,983	3,071
19	29,974	38,506	1,088	1,998	3,086
20	30,120	38,886	1,088	1,942	3,030
Avg	30,161	38,745	1,088	1,985	3,073
Stddev	134	281	0	24	24

Table 11. Wallace Tree 4 LP SBF Partition Simulation Results

Circuit	- Wallace
Partition	- SBF
Num_Vertices	- 1,050
Num_Arcs	- 1,770
Num_LPs	- 4
Inter-LP Arcs	- 346
Wght_Inter_LP_Arcs	- 346.0
Avg_Wght_Arcs	- 86.5
Stddev_Wght_Out_Arcs	- 56.9
Maxdev_Wght_Out_Arcs	- 75.5
Stddev_Wght_In_Arcs	- 53.3
Maxdev_Wght_In_Arcs	- 41.5
Comm_Cost_Factor	- 19.55%
Comm_Dist_Factor	- 87.28%
LP_Output_Lines	- 9
Lookahead_Factor	- 0.783
Avg_Comp_Load	- 262.5
Stddev_Comp_Load	- 0.6
Maxdev_Comp_Load	- 0.5
Load_Delta_Factor	- 0.19%
Predicted Speedup	- 3.12

Trial	Sim Time (ms)	Total Time (ms)	Reals Sent	Nulls Sent	Total Sent
1	20,702	29,879	2,785	10,088	12,873
2	19,679	28,960	2,785	10,277	13,062
3	19,751	29,064	2,785	10,186	12,971
4	19,560	28,860	2,785	10,070	12,855
5	19,622	28,899	2,785	10,168	12,953
6	20,498	29,605	2,785	10,368	13,153
7	19,562	28,883	2,785	10,201	12,986
8	19,885	29,089	2,785	10,224	13,009
9	19,815	29,133	2,785	10,129	12,914
10	20,302	29,457	2,785	10,126	12,911
11	19,980	29,294	2,785	10,129	12,914
12	19,613	28,720	2,785	10,142	12,927
13	19,891	29,117	2,785	10,221	13,006
14	19,377	28,676	2,785	10,305	13,090
15	19,442	28,815	2,785	10,177	12,962
16	19,701	29,080	2,785	10,231	13,016
17	19,556	28,705	2,785	10,122	12,907
18	19,526	28,855	2,785	10,139	12,924
19	20,032	29,205	2,785	10,189	12,974
20	20,153	29,443	2,785	10,111	12,896
Avg	19,832	29,087	2,785	10,180	12,965
Stddev	347	312	0	73	73

Table 12. Wallace Tree 8 LP SBF Partition Simulation Results

Circuit	- Wallace
Partition	- SBF
Num_Vertices	- 1,050
Num_Arcs	- 1,770
Num_LPs	- 8
Inter-LP Arcs	- 550
Wght_Inter_LP_Arcs	- 550.0
Avg_Wght_Arcs	- 68.8
Stddev_Wght_Out_Arcs	- 34.5
Maxdev_Wght_Out_Arcs	- 76.3
Stddev_Wght_In_Arcs	- 24.3
Maxdev_Wght_In_Arcs	- 23.3
Comm_Cost_Factor	- 31.07%
Comm_Dist_Factor	- 110.91%
LP_Output_Lines	- 29
Lookahead_Factor	- 0.690
Avg_Comp_Load	- 131.3
Stddev_Comp_Load	- 0.5
Maxdev_Comp_Load	- 0.8
Load_Delta_Factor	- 0.57%
Predicted Speedup	- 3.20

Trial	Sim Time (ms)	Total Time (ms)	Reals Sent	Nulls Sent	Total Sent
1	22,898	34,270	4,762	31,099	35,861
2	23,176	34,433	4,762	30,994	35,756
3	23,409	34,884	4,762	31,212	35,974
4	23,417	34,826	4,762	30,948	35,710
5	24,514	35,917	4,762	31,007	35,769
6	24,146	35,407	4,762	30,801	35,563
7	23,983	35,318	4,762	30,754	35,516
8	24,280	35,639	4,762	30,618	35,380
9	23,605	34,918	4,762	30,796	35,558
10	24,409	35,518	4,762	30,818	35,580
11	24,595	36,192	4,762	30,952	35,714
12	24,177	35,652	4,762	30,904	35,666
13	23,672	34,972	4,762	30,938	35,700
14	23,870	35,257	4,762	30,992	35,754
15	22,956	34,467	4,762	30,937	35,699
16	22,488	34,074	4,762	30,945	35,707
17	24,136	35,651	4,762	30,463	35,225
18	24,438	35,652	4,762	30,933	35,695
19	23,272	34,660	4,762	30,360	35,122
20	23,198	34,756	4,762	30,604	35,366
Avg	23,732	35,123	4,762	30,854	35,616
Stddev	596	569	0	204	204

Table 13. Wallace Tree 2 LP AB2 Partition Simulation Results

Circuit	- Wallace
Partition	- AB2_2
Num_Vertices	- 1,050
Num_Arcs	- 1,770
Num_LPs	- 2
Inter-LP Arcs	- 97
Wght_Inter_LP_Arcs	- 97.0
Avg_Wght_Arcs	- 48.5
Stddev_Wght_Out_Arcs	- 24.7
Maxdev_Wght_Out_Arcs	- 17.5
Stddev_Wght_In_Arcs	- 24.7
Maxdev_Wght_In_Arcs	- 17.5
Comm_Cost_Factor	- 5.48%
Comm_Dist_Factor	- 36.08%
LP_Output_Lines	- 2
Lookahead_Factor	- 1.000
Avg_Comp_Load	- 525.0
Stddev_Comp_Load	- 1.4
Maxdev_Comp_Load	- 1.0
Load_Delta_Factor	- 0.19%
Predicted Speedup	- 1.94

Trial	Sim Time (ms)	Total Time (ms)	Reals Sent	Nulls Sent	Total Sent
1	27,422	35,890	1,045	2,216	3,261
2	27,472	35,935	1,045	2,214	3,259
3	27,453	35,927	1,045	2,218	3,263
4	27,373	35,852	1,045	2,210	3,255
5	27,451	35,933	1,045	2,217	3,262
6	27,456	35,926	1,045	2,216	3,261
7	27,457	35,931	1,045	2,216	3,261
8	27,461	35,927	1,045	2,217	3,262
9	27,635	36,987	1,045	2,212	3,257
10	27,486	36,869	1,045	2,216	3,261
11	27,454	35,945	1,045	2,217	3,262
12	27,460	37,337	1,045	2,216	3,261
13	27,456	35,935	1,045	2,220	3,265
14	27,448	35,938	1,045	2,216	3,261
15	27,451	35,939	1,045	2,217	3,262
16	27,457	35,934	1,045	2,217	3,262
17	27,450	35,936	1,045	2,221	3,266
18	27,454	35,929	1,045	2,215	3,260
19	27,444	35,920	1,045	2,217	3,262
20	27,461	35,961	1,045	2,217	3,262
Avg	27,460	36,098	1,045	2,216	3,261
Stddev	45	414	0	2	2

Table 14. Wallace Tree 4 LP AB2 Partition Simulation Results

Circuit	- Wallace
Partition	- AB2_2
Num_Vertices	- 1,050
Num_Arcs	- 1,770
Num_LPs	- 4
Inter-LP Arcs	- 192
Wght_Inter_LP_Arcs	- 192.0
Avg_Wght_Arcs	- 48.0
Stddev_Wght_Out_Arcs	- 39.6
Maxdev_Wght_Out_Arcs	- 58.0
Stddev_Wght_In_Arcs	- 32.7
Maxdev_Wght_In_Arcs	- 27.0
Comm_Cost_Factor	- 10.85%
Comm_Dist_Factor	- 120.83%
LP_Output_Lines	- 9
Lookahead_Factor	- 0.514
Avg_Comp_Load	- 262.5
Stddev_Comp_Load	- 4.4
Maxdev_Comp_Load	- 3.5
Load_Delta_Factor	- 1.33%
Predicted Speedup	- 3.16

Trial	Sim Time (ms)	Total Time (ms)	Reals Sent	Nulls Sent	Total Sent
1	19,337	28,403	1,760	8,710	10,470
2	19,630	28,800	1,760	8,730	10,490
3	18,695	27,939	1,760	8,814	10,574
4	19,451	28,601	1,760	8,800	10,560
5	20,071	29,037	1,760	8,845	10,605
6	19,669	28,912	1,760	8,795	10,555
7	18,968	28,068	1,760	8,731	10,491
8	18,780	28,000	1,760	8,733	10,493
9	19,695	28,931	1,760	8,799	10,559
10	20,038	29,015	1,760	8,739	10,499
11	20,124	29,310	1,760	8,583	10,343
12	20,071	29,151	1,760	8,628	10,388
13	19,736	28,795	1,760	8,750	10,510
14	19,817	28,924	1,760	8,785	10,545
15	18,937	28,091	1,760	8,792	10,552
16	19,818	28,889	1,760	8,731	10,491
17	20,076	29,134	1,760	8,731	10,491
18	18,763	27,876	1,760	8,860	10,620
19	19,064	28,287	1,760	8,723	10,483
20	20,193	29,279	1,760	8,691	10,451
Avg	19,547	28,672	1,760	8,749	10,509
Stddev	498	463	0	65	65

Table 15. Wallace Tree 8 LP AB2 Partition Simulation Results

Circuit	- Wallace
Partition	- AB2_2
Num_Vertices	- 1,050
Num_Arcs	- 1,770
Num_LPs	- 8
Inter-LP Arcs	- 295
Wght_Inter_LP_Arcs	- 295.0
Avg_Wght_Arcs	- 36.9
Stddev_Wght_Out_Arcs	- 31.6
Maxdev_Wght_Out_Arcs	- 71.1
Stddev_Wght_In_Arcs	- 23.4
Maxdev_Wght_In_Arcs	- 39.1
Comm_Cost_Factor	- 16.67%
Comm_Dist_Factor	- 192.88%
LP_Output_Lines	- 26
Lookahead_Factor	- 0.547
Avg_Comp_Load	- 131.3
Stddev_Comp_Load	- 3.5
Maxdev_Comp_Load	- 1.8
Load_Delta_Factor	- 1.33%
Predicted Speedup	- 3.99

Trial	Sim Time (ms)	Total Time (ms)	Reals Sent	Nulls Sent	Total Sent
1	17,368	27,974	2,762	18,649	21,411
2	17,981	28,784	2,762	18,805	21,567
3	16,555	27,159	2,762	18,981	21,743
4	16,516	27,351	2,762	18,935	21,697
5	17,058	27,887	2,762	18,901	21,663
6	17,251	28,176	2,762	18,983	21,745
7	18,194	29,058	2,762	18,882	21,644
8	18,514	29,377	2,762	18,944	21,706
9	17,060	27,974	2,762	18,916	21,678
10	17,019	27,931	2,762	18,914	21,676
11	19,632	30,446	2,762	18,808	21,570
12	17,822	28,780	2,762	19,051	21,813
13	16,303	26,992	2,762	18,865	21,627
14	16,803	27,320	2,762	18,762	21,524
15	19,024	29,928	2,762	18,886	21,648
16	16,403	27,237	2,762	18,972	21,734
17	17,583	28,825	2,762	18,778	21,540
18	17,987	28,829	2,762	18,855	21,617
19	16,710	27,691	2,762	18,816	21,578
20	17,181	27,985	2,762	19,059	21,821
Avg	17,448	28,285	2,762	18,888	21,650
Stddev	875	922	0	98	98

Table 16. Associative Memory 1 LP Simulation Results

Circuit	- Assoc_Mem
Partition	- Random
Num_Vertices	- 4,343
Num_Arcs	- 9,312
Num_LPs	- 1
Inter-LP Arcs	- 0
Wght_Inter_LP_Arcs	- 0.0
Avg_Wght_Arcs	- 0.0
Stddev_Wght_Out_Arcs	- 0.0
Maxdev_Wght_Out_Arcs	- 0.0
Stddev_Wght_In_Arcs	- 0.0
Maxdev_Wght_In_Arcs	- 0.0
Comm_Cost_Factor	- 0.00%
Comm_Dist_Factor	- 0.00%
LP_Output_Lines	- 0
Lookahead_Factor	- 0.000
Avg_Comp_Load	- 4,343.0
Stddev_Comp_Load	- 0.0
Maxdev_Comp_Load	- 0.0
Load_Delta_Factor	- 0.00%
Predicted Speedup	- 0.00

Trial	Sim Time(ms)	Total Time(ms)	Reals Sent	Nulls Sent	Total Sent
1	4,380,494	4,552,528	0	0	0
2	4,358,019	4,530,100	0	0	0
3	4,358,013	4,530,114	0	0	0
4	4,532,362	4,704,383	0	0	0
5	4,381,789	4,553,802	0	0	0
6	4,358,020	4,530,090	0	0	0
7	4,358,017	4,530,032	0	0	0
8	4,358,017	4,530,028	0	0	0
9	4,357,986	4,530,135	0	0	0
10	4,358,022	4,530,113	0	0	0
Avg	4,388,074	4,552,127	0	0	0
Stddev	51,577	51,563	0	0	0

Table 17. Associative Memory 2 LP Random Partition Simulation Results

Circuit	-	Assoc_Mem
Partition	-	Random
Num_Vertices	-	4,243
Num_Arcs	-	9,312
Num_LPs	-	2
Inter-LP Arcs	-	4,657
Wght_Inter_LP_Arcs	-	4,657.0
Avg_Wght_Arcs	-	2,328.5
Stddev_Wght_Out_Arcs	-	7.8
Maxdev_Wght_Out_Arcs	-	5.5
Stddev_Wght_In_Arcs	-	7.8
Maxdev_Wght_In_Arcs	-	5.5
Comm_Cost_Factor	-	50.01%
Comm_Dist_Factor	-	0.24%
LP_Output_Lines	-	2
Lookahead_Factor	-	1.000
Avg_Comp_Load	-	2,121.5
Stddev_Comp_Load	-	0.7
Maxdev_Comp_Load	-	0.5
Load_Delta_Factor	-	0.02%
Predicted Speedup	-	1.95

Trial	Sim Time(ms)	Total Time(ms)	Reals Sent	Nulls Sent	Total Sent
1	1,782,224	1,940,390	337,825	13,396	351,221
2	1,820,868	1,979,000	337,825	13,403	351,228
3	1,821,686	1,979,816	337,825	13,357	351,182
4	1,836,876	1,995,004	337,825	13,490	351,315
5	1,833,512	1,991,699	337,825	13,432	351,257
6	1,815,789	1,973,916	337,825	13,473	351,298
7	1,776,878	1,935,006	337,825	13,524	351,349
8	1,774,200	1,932,324	337,825	13,390	351,215
9	1,910,705	2,068,833	337,825	13,530	351,355
10	1,882,788	2,040,916	337,825	13,452	351,277
Avg	1,825,553	1,983,690	337,825	13,445	351,270
Stddev	42,112	42,110	0	56	56

Table 18. Associative Memory 4 LP Random Partition Simulation Results

Circuit	-	Assoc_Mem
Partition	-	Random
Num_Vertices	-	4,243
Num_Arcs	-	9,312
Num_LPs	-	4
Inter-LP Arcs	-	6,961
Wght_Inter_LP_Arcs	-	6,961.0
Avg_Wght_Arcs	-	1,740.3
Stddev_Wght_Out_Arcs	-	317.9
Maxdev_Wght_Out_Arcs	-	472.8
Stddev_Wght_In_Arcs	-	91.0
Maxdev_Wght_In_Arcs	-	50.8
Comm_Cost_Factor	-	74.75%
Comm_Dist_Factor	-	27.17%
LP_Output_Lines	-	12
Lookahead_Factor	-	1.000
Avg_Comp_Load	-	1,060.8
Stddev_Comp_Load	-	0.5
Maxdev_Comp_Load	-	0.3
Load_Delta_Factor	-	0.02%
Predicted Speedup	-	3.11

Trial	Sim Time(ms)	Total Time(ms)	Reals Sent	Nulls Sent	Total Sent
1	1,339,947	1,496,103	608,106	93,793	701,899
2	1,331,954	1,488,188	608,106	93,842	701,948
3	1,352,230	1,508,436	608,106	93,677	701,783
4	1,283,714	1,439,881	608,104	94,001	702,105
5	1,395,191	1,551,495	608,104	92,901	701,005
6	1,366,912	1,523,087	608,108	93,300	701,408
7	1,331,230	1,487,421	608,104	93,758	701,862
8	1,307,684	1,463,875	608,104	93,751	701,855
9	1,348,794	1,504,971	608,104	93,457	701,561
10	1,367,435	1,523,614	608,106	93,312	701,418
Avg	1,342,569	1,498,707	608,105	93,579	701,684
Stddev	30,068	30,089	1	314	314

Table 19. Associative Memory 8 LP Random Partition Simulation Results

Circuit	-	Assoc_Mem
Partition	-	Random
Num_Vertices	-	4,243
Num_Arcs	-	9,312
Num_LPs	-	8
Inter-LP Arcs	-	8,129
Wght_Inter_LP_Arcs	-	8,129.0
Avg_Wght_Arcs	-	1,016.1
Stddev_Wght_Out_Arcs	-	210.3
Maxdev_Wght_Out_Arcs	-	458.9
Stddev_Wght_In_Arcs	-	35.6
Maxdev_Wght_In_Arcs	-	55.9
Comm_Cost_Factor	-	87.30%
Comm_Dist_Factor	-	45.16%
LP_Output_Lines	-	56
Lookahead_Factor	-	1.000
Avg_Comp_Load	-	530.4
Stddev_Comp_Load	-	0.5
Maxdev_Comp_Load	-	0.6
Load_Delta_Factor	-	0.12%
Predicted Speedup	-	2.88

Trial	Sim Time(ms)	Total Time(ms)	Reals Sent	Nulls Sent	Total Sent
1	1,399,473	1,553,999	753,718	444,058	1,197,776
2	1,437,316	1,591,823	753,712	444,251	1,197,963
3	1,453,001	1,607,432	753,712	443,595	1,197,307
4	1,460,823	1,615,389	753,720	443,856	1,197,576
5	1,507,919	1,662,466	753,716	443,898	1,197,614
6	1,428,524	1,582,931	753,720	443,991	1,197,711
7	1,468,822	1,623,349	753,718	443,887	1,197,605
8	1,447,797	1,602,330	753,716	443,781	1,197,497
9	1,455,769	1,610,297	753,718	443,822	1,197,540
10	1,415,692	1,570,238	753,714	443,750	1,197,464
Avg	1,447,514	1,602,025	753,716	443,889	1,197,605
Stddev	28,572	28,583	3	171	172

Table 20. Associative Memory 2 LP SDF Partition Simulation Results

Circuit	- Assoc_Mem
Partition	- SDF
Num_Vertices	- 4,243
Num_Arcs	- 9,312
Num_LPs	- 2
Inter-LP Arcs	- 1,601
Wght_Inter_LP_Arcs	- 1,601.0
Avg_Wght_Arcs	- 800.5
Stddev_Wght_Out_Arcs	- 878.9
Maxdev_Wght_Out_Arcs	- 621.5
Stddev_Wght_In_Arcs	- 878.9
Maxdev_Wght_In_Arcs	- 621.5
Comm_Cost_Factor	- 17.19%
Comm_Dist_Factor	- 77.64%
LP_Output_Lines	- 2
Lookahead_Factor	- 0.667
Avg_Comp_Load	- 2,121.5
Stddev_Comp_Load	- 0.7
Maxdev_Comp_Load	- 0.5
Load_Delta_Factor	- 0.02%
Predicted Speedup	- 1.97

Trial	Sim Time(ms)	Total Time(ms)	Reals Sent	Nulls Sent	Total Sent
1	1,746,422	1,910,040	25,476	7,882	33,358
2	1,709,357	1,873,118	25,476	7,788	33,264
3	1,689,775	1,853,317	25,476	7,798	33,274
4	1,714,009	1,877,561	25,476	7,769	33,245
5	1,747,851	1,911,490	25,476	7,878	33,354
6	1,739,992	1,903,528	25,476	7,880	33,356
7	1,704,852	1,868,399	25,476	7,875	33,351
8	1,729,874	1,893,441	25,476	7,878	33,354
9	1,674,304	1,837,858	25,476	7,878	33,354
10	1,692,469	1,856,017	25,476	7,859	33,335
Avg	1,714,891	1,878,477	25,476	7,849	33,325
Stddev	24,183	24,199	0	43	43

Table 21. Associative Memory 4 LP SDF Partition Simulation Results

Circuit	- Assoc_Mem
Partition	- SDF
Num_Vertices	- 4,243
Num_Arcs	- 9,312
Num_LPs	- 4
Inter-LP Arcs	- 2,769
Wght_Inter_LP_Arcs	- 2,769.0
Avg_Wght_Arcs	- 692.3
Stddev_Wght_Out_Arcs	- 1,175.0
Maxdev_Wght_Out_Arcs	- 1,759.8
Stddev_Wght_In_Arcs	- 331.1
Maxdev_Wght_In_Arcs	- 242.8
Comm_Cost_Factor	- 29.74%
Comm_Dist_Factor	- 254.21%
LP_Output_Lines	- 11
Lookahead_Factor	- 0.579
Avg_Comp_Load	- 1,060.8
Stddev_Comp_Load	- 0.5
Maxdev_Comp_Load	- 0.3
Load_Delta_Factor	- 0.02%
Predicted Speedup	- 3.25

Trial	Sim Time(ms)	Total Time(ms)	Reals Sent	Nulls Sent	Total Sent
1	1,018,731	1,175,324	43,736	63,671	107,407
2	1,244,913	1,401,463	43,736	62,766	106,502
3	969,530	1,126,128	43,736	58,552	102,288
4	960,683	1,117,149	43,736	58,544	102,280
5	1,082,135	1,238,733	43,736	60,483	104,219
6	1,111,279	1,267,666	43,736	60,495	104,231
7	1,074,745	1,231,118	43,736	60,608	104,344
8	1,078,213	1,234,549	43,736	60,379	104,115
9	1,098,268	1,254,808	43,736	60,544	104,280
10	1,068,414	1,224,865	43,736	60,713	104,449
Avg	1,070,691	1,227,180	43,736	60,676	104,412
Stddev	76,161	76,153	0	1,503	1,503

Table 22. Associative Memory 8 LP SDF Partition Simulation Results

Circuit	- Assoc_Mem
Partition	- SDF
Num_Vertices	- 4,243
Num_Arcs	- 9,312
Num_LPs	- 8
Inter-LP Arcs	- 3,145
Wght_Inter_LP_Arcs	- 3,145.0
Avg_Wght_Arcs	- 393.1
Stddev_Wght_Out_Arcs	- 595.4
Maxdev_Wght_Out_Arcs	- 1,309.9
Stddev_Wght_In_Arcs	- 108.9
Maxdev_Wght_In_Arcs	- 86.9
Comm_Cost_Factor	- 33.77%
Comm_Dist_Factor	- 333.20%
LP_Output_Lines	- 44
Lookahead_Factor	- 0.506
Avg_Comp_Load	- 530.4
Stddev_Comp_Load	- 0.5
Maxdev_Comp_Load	- 0.6
Load_Delta_Factor	- 0.12%
Predicted Speedup	- 3.61

Trial	Sim Time(ms)	Total Time(ms)	Reals Sent	Nulls Sent	Total Sent
1	1,252,477	1,407,312	77,148	260,049	337,197
2	1,335,871	1,490,746	77,148	257,658	334,806
3	1,239,755	1,394,570	77,148	257,880	335,028
4	1,246,327	1,401,288	77,148	257,904	335,052
5	1,282,282	1,437,108	77,148	257,251	334,399
6	1,275,896	1,430,781	77,148	256,566	333,714
7	1,140,868	1,295,779	77,148	257,213	334,361
8	1,174,754	1,329,534	77,148	256,785	333,933
9	1,216,161	1,371,007	77,148	257,451	334,599
10	1,237,724	1,392,678	77,148	257,394	334,542
Avg	1,240,212	1,395,000	77,148	257,615	334,763
Stddev	52,077	52,080	0	908	908

Table 23. Associative Memory 2 LP SBF Partition Simulation Results

Circuit	-	Assoc_Mem	Trial	Sim Time(ms)	Total Time(ms)	Reals Sent	Nulls Sent	Total Sent
			1	2,539,612	2,701,319	264,131	7,038	271,169
Partition	-	SBF	2	2,611,170	2,772,921	264,131	7,070	271,201
Num_Vertices	-	4,243	3	2,658,448	2,820,698	264,131	7,037	271,168
Num_Arcs	-	9,312	4	2,554,856	2,721,239	264,131	7,056	271,187
Num_LPs	-	2	5	2,556,957	2,745,318	264,131	7,045	271,176
Inter-LP Arcs	-	3,707	6	2,623,437	2,811,369	264,131	7,054	271,185
Wght_Inter_LP_Arcs	-	3,707.0	7	2,633,263	2,782,405	264,131	7,068	271,199
Avg_Wght_Arcs	-	1,859.5	8	2,612,455	2,812,482	264,131	7,056	271,187
Stddev_Wght_Out_Arcs	-	672.5	9	2,589,795	2,742,364	264,131	7,081	271,212
Maxdev_Wght_Out_Arcs	-	475.5	10	2,578,644	2,733,265	264,131	7,069	271,200
Stddev_Wght_In_Arcs	-	672.5						
Maxdev_Wght_In_Arcs	-	475.5						
Comm_Cost_Factor	-	39.81%						
Comm_Dist_Factor	-	25.65%						
LP_Output_Lines	-	2						
Lookahead_Factor	-	0.667						
Avg_Comp_Load	-	2,121.5						
Stddev_Comp_Load	-	0.7						
Maxdev_Comp_Load	-	0.5						
Load_Delta_Factor	-	0.02%						
Predicted Speedup	-	1.96	Avg	2,595,864	2,764,338	264,131	7,057	271,188
			Stddev	36,439	39,680	0	14	14

Table 24. Associative Memory 4 LP SBF Partition Simulation Results

Circuit	-	Assoc_Mem	Trial	Sim Time(ms)	Total Time(ms)	Reals Sent	Nulls Sent	Total Sent
			1	1,676,393	1,830,694	446,656	86,113	532,769
Partition	-	SBF	2	1,802,239	1,959,591	446,656	85,997	532,653
Num_Vertices	-	4,243	3	1,761,017	1,915,298	446,656	85,798	532,454
Num_Arcs	-	9,312	4	1,900,192	2,054,454	446,656	86,480	533,136
Num_LPs	-	4	5	1,923,823	2,078,081	446,656	86,484	533,140
Inter-LP Arcs	-	5,967	6	1,763,242	1,917,510	446,656	86,370	533,026
Wght_Inter_LP_Arcs	-	5,967.0	7	1,761,027	1,915,316	446,656	85,951	532,607
Avg_Wght_Arcs	-	1,491.8	8	1,900,213	2,054,511	446,656	86,418	533,074
Stddev_Wght_Out_Arcs	-	927.1	9	1,923,801	2,078,057	446,656	86,496	533,152
Maxdev_Wght_Out_Arcs	-	1,312.3	10	1,763,237	1,917,487	446,656	86,397	533,053
Stddev_Wght_In_Arcs	-	661.9						
Maxdev_Wght_In_Arcs	-	804.3						
Comm_Cost_Factor	-	64.08%						
Comm_Dist_Factor	-	87.97%						
LP_Output_Lines	-	12						
Lookahead_Factor	-	0.600						
Avg_Comp_Load	-	1,060.8						
Stddev_Comp_Load	-	0.5						
Maxdev_Comp_Load	-	0.3						
Load_Delta_Factor	-	0.02%						
Predicted Speedup	-	3.10	Avg	1,817,518	1,972,100	446,656	86,250	532,906
			Stddev	82,883	82,824	0	247	247

Table 25. Associative Memory 8 LP SBF Partition Simulation Results

Circuit	-	Assoc_Mem	Trial	Sim Time(ms)	Total Time(ms)	Reals Sent	Nulls Sent	Total Sent
			1	2,464,665	2,619,507	501,074	294,435	795,509
Partition	-	SBF	2	2,095,664	2,250,030	501,072	295,665	796,737
Num_Vertices	-	4,243	3	2,398,870	2,552,531	501,070	295,631	796,701
Num_Arcs	-	9,312	4	2,384,139	2,538,588	501,070	296,313	797,383
Num_LPs	-	8	5	2,428,855	2,583,033	501,070	295,721	796,791
Inter-LP Arcs	-	7,257	6	2,392,312	2,546,995	501,072	295,645	796,717
Wght_Inter_LP_Arcs	-	7,257.0	7	2,233,466	2,387,701	501,070	296,257	797,327
Avg_Wght_Arcs	-	907.1	8	2,284,345	2,546,995	501,070	296,347	797,417
Stddev_Wght_Out_Arcs	-	878.9	9	2,384,786	2,551,485	501,070	295,783	796,853
Maxdev_Wght_Out_Arcs	-	2,108.9	10	2,236,645	2,430,125	501,072	296,224	797,296
Stddev_Wght_In_Arcs	-	400.3						
Maxdev_Wght_In_Arcs	-	475.9						
Comm_Cost_Factor	-	77.93%						
Comm_Dist_Factor	-	232.48%						
LP_Output_Lines	-	39						
Lookahead_Factor	-	0.542						
Avg_Comp_Load	-	530.4						
Stddev_Comp_Load	-	0.5						
Maxdev_Comp_Load	-	0.6						
Load_Delta_Factor	-	0.12%						
Predicted Speedup	-	2.71	Avg	2,330,375	2,500,699	501,071	295,802	796,873
			Stddev	108,395	106,050	1	538	537

Table 26. Associative Memory 2 LP AB1 Partition Simulation Results

Circuit	-	Assoc_Mem	Trial	Sim Time(ms)	Total Time(ms)	Reals Sent	Nulls Sent	Total Sent
			1	1,723,325	1,886,124	25,058	7,993	33,051
Partition	-	AB1_7	2	1,736,776	1,899,575	25,058	7,991	33,049
Num_Vertices	-	4,243	3	1,701,738	1,864,537	25,058	7,925	32,983
Num_Arcs	-	9,312	4	1,706,377	1,869,186	25,058	8,076	33,134
Num_LPs	-	2	5	1,713,874	1,876,696	25,058	8,081	33,139
Inter-LP Arcs	-	1,274	6	1,764,724	1,927,553	25,058	8,077	33,135
Wght_Inter_LP_Arcs	-	1,274.0	7	1,709,578	1,872,402	25,058	8,054	33,112
Avg_Wght_Arcs	-	637.0	8	1,709,056	1,871,873	25,058	8,044	33,102
Stddev_Wght_Out_Arcs	-	362.0	9	1,734,024	1,896,854	25,058	8,067	33,125
Maxdev_Wght_Out_Arcs	-	256.0	10	1,671,783	1,834,610	25,058	8,152	33,210
Stddev_Wght_In_Arcs	-	362.0						
Maxdev_Wght_In_Arcs	-	256.0						
Comm_Cost_Factor	-	13.68%						
Comm_Dist_Factor	-	40.19%						
LP_Output_Lines	-	2						
Lookahead_Factor	-	0.667						
Avg_Comp_Load	-	2,121.5						
Stddev_Comp_Load	-	14.8						
Maxdev_Comp_Load	-	10.5						
Load_Delta_Factor	-	0.49%						
Predicted Speedup	-	1.96	Avg	1,717,126	1,879,941	25,058	8,046	33,104
			Stddev	23,488	23,488	0	59	59

Table 27. Associative Memory 4 LP AB1 Partition Simulation Results

Circuit	-	Assoc_Mem	Trial	Sim Time(ms)	Total Time(ms)	Reals Sent	Nulls Sent	Total Sent
			1	1,103,822	1,260,109	52,043	47,559	99,602
Partition	-	AB1_7	2	1,006,145	1,162,880	52,043	49,737	101,780
Num_Vertices	-	4,243	3	969,624	1,126,411	52,043	46,895	98,938
Num_Arcs	-	9,312	4	991,067	1,147,950	52,043	47,094	99,137
Num_LPs	-	4	5	899,179	1,055,956	52,043	47,783	99,826
Inter-LP Arcs	-	1,920	6	896,519	1,053,360	52,043	47,090	99,133
Wght_Inter_LP_Arcs	-	1,920.0	7	1,086,595	1,242,699	52,043	47,881	99,924
Avg_Wght_Arcs	-	480.0	8	1,016,290	1,173,655	52,043	46,779	98,822
Stddev_Wght_Out_Arcs	-	670.9	9	943,176	1,100,039	52,043	47,700	99,743
Maxdev_Wght_Out_Arcs	-	1,005.0	10	1,041,043	1,197,878	52,043	47,048	99,091
Stddev_Wght_In_Arcs	-	76.9						
Maxdev_Wght_In_Arcs	-	91.0						
Comm_Cost_Factor	-	20.62%						
Comm_Dist_Factor	-	209.38%						
LP_Output_Lines	-	9						
Lookahead_Factor	-	0.692						
Avg_Comp_Load	-	1,060.8						
Stddev_Comp_Load	-	10.5						
Maxdev_Comp_Load	-	5.3						
Load_Delta_Factor	-	0.49%						
Predicted Speedup	-	3.60	Avg	995,346	1,152,094	52,043	47,557	99,600
			Stddev	67,196	67,031	0	816	816

Table 28. Associative Memory 8 LP AB1 Partition Simulation Results

Circuit	-	Assoc_Mem	Trial	Sim Time(ms)	Total Time(ms)	Reals Sent	Nulls Sent	Total Sent
			1	1,186,793	1,342,304	82,548	232,155	314,703
Partition	-	AB1_7	2	1,169,894	1,326,063	82,548	232,437	314,985
Num_Vertices	-	4,243	3	1,111,823	1,268,388	82,548	232,051	314,599
Num_Arcs	-	9,312	4	1,185,544	1,342,227	82,548	232,693	315,241
Num_LPs	-	8	5	1,216,265	1,375,978	82,548	232,332	314,880
Inter-LP Arcs	-	2,594	6	1,136,925	1,293,550	82,548	232,734	315,282
Wght_Inter_LP_Arcs	-	2,594.0	7	1,120,123	1,276,639	82,548	232,716	315,264
Avg_Wght_Arcs	-	324.3	8	1,114,596	1,281,187	82,548	232,627	315,175
Stddev_Wght_Out_Arcs	-	364.1	9	1,184,460	1,341,099	82,548	232,007	314,555
Maxdev_Wght_Out_Arcs	-	585.8	10	1,149,380	1,305,815	82,548	232,309	314,857
Stddev_Wght_In_Arcs	-	38.5						
Maxdev_Wght_In_Arcs	-	70.8						
Comm_Cost_Factor	-	27.86%						
Comm_Dist_Factor	-	180.65%						
LP_Output_Lines	-	40						
Lookahead_Factor	-	0.602						
Avg_Comp_Load	-	530.4						
Stddev_Comp_Load	-	7.4						
Maxdev_Comp_Load	-	2.6						
Load_Delta_Factor	-	0.49%						
Predicted Speedup	-	5.40	Avg	1,158,580	1,315,325	82,548	232,406	314,954
			Stddev	33,199	33,674	0	264	264

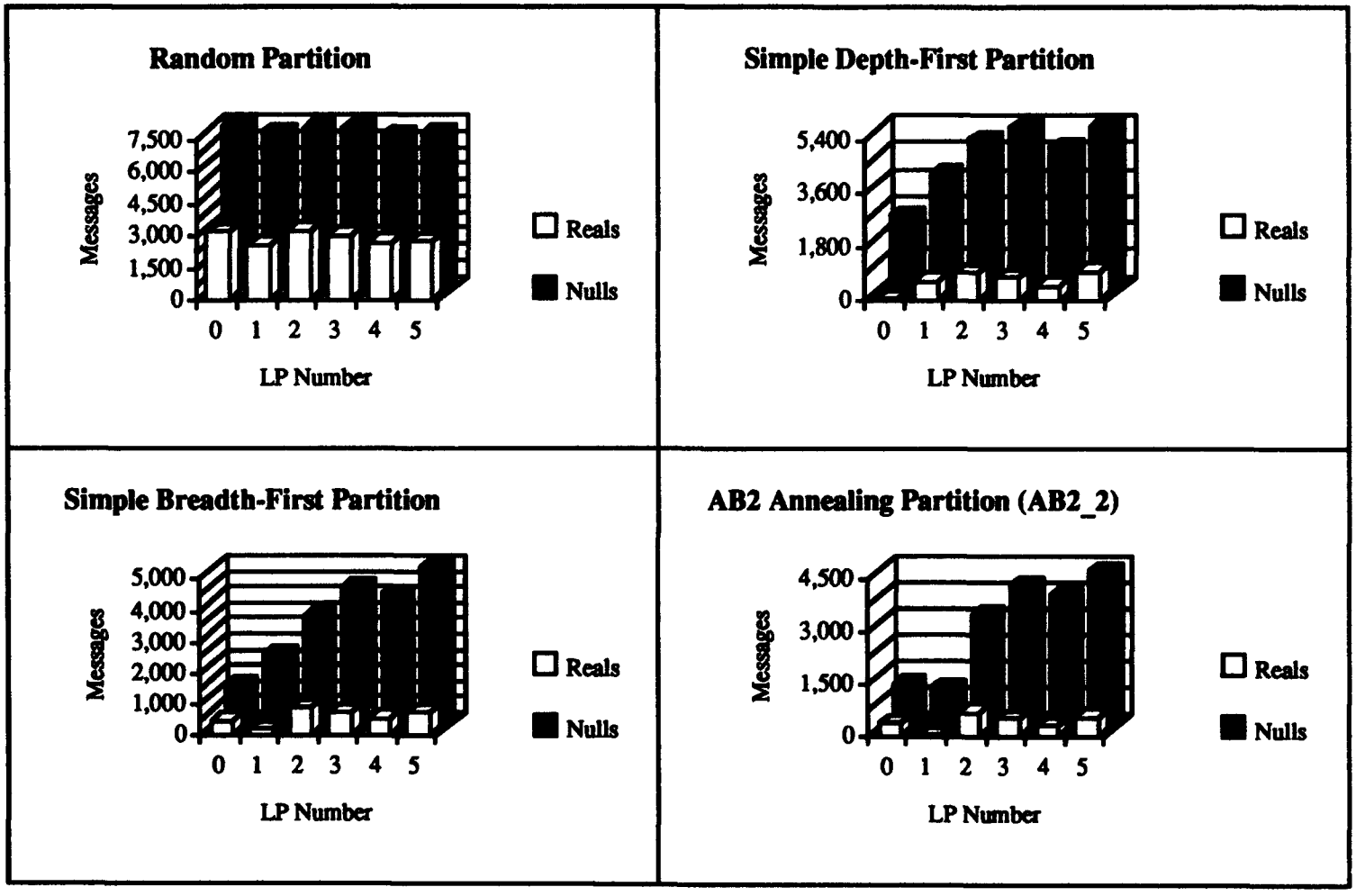


Figure 75. Wallace Tree 6 LP Reals Sent vs. Nulls Sent Message Analysis

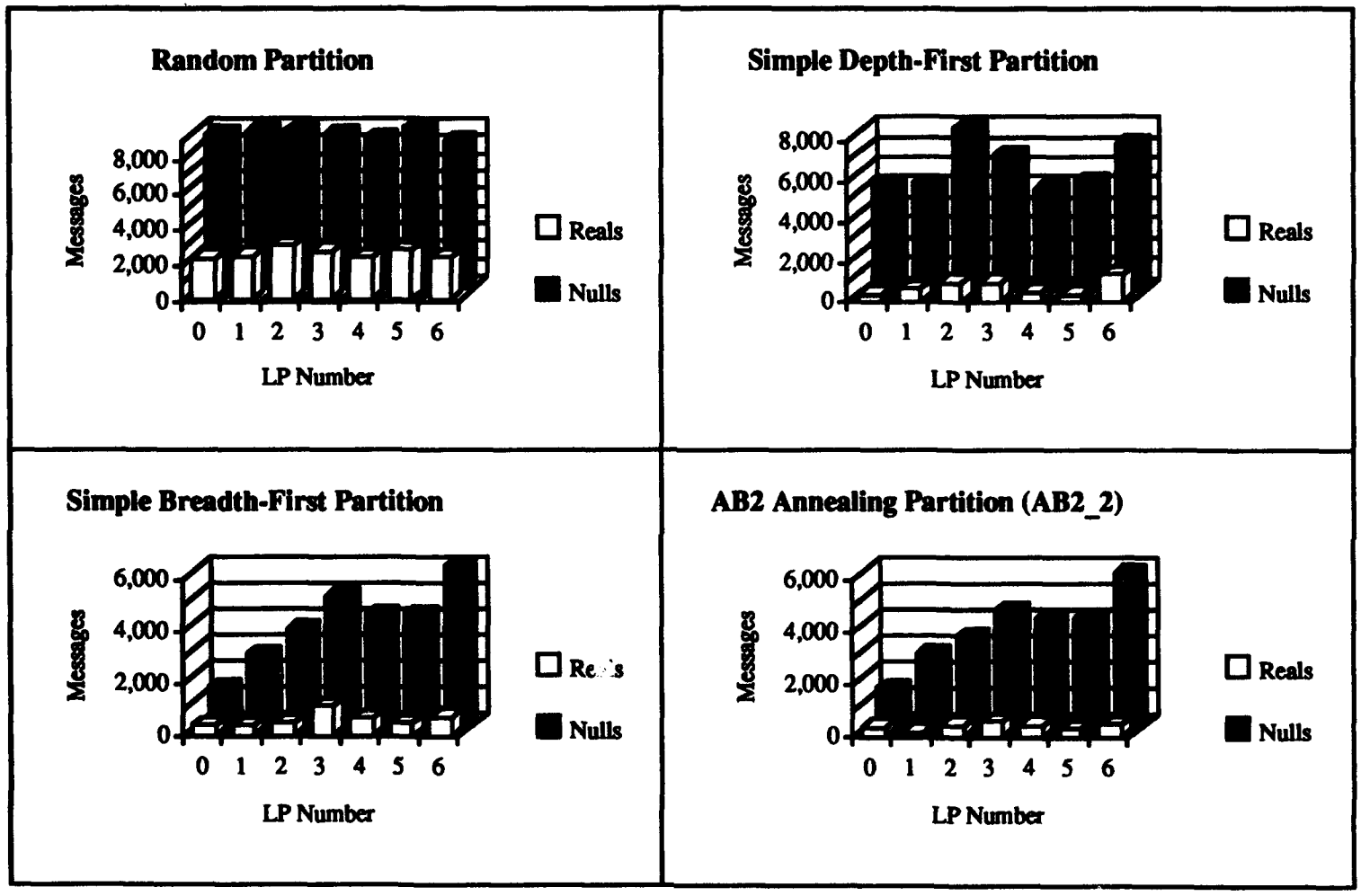


Figure 76. Wallace Tree 7 LP Reals Sent vs. Nulls Sent Message Analysis

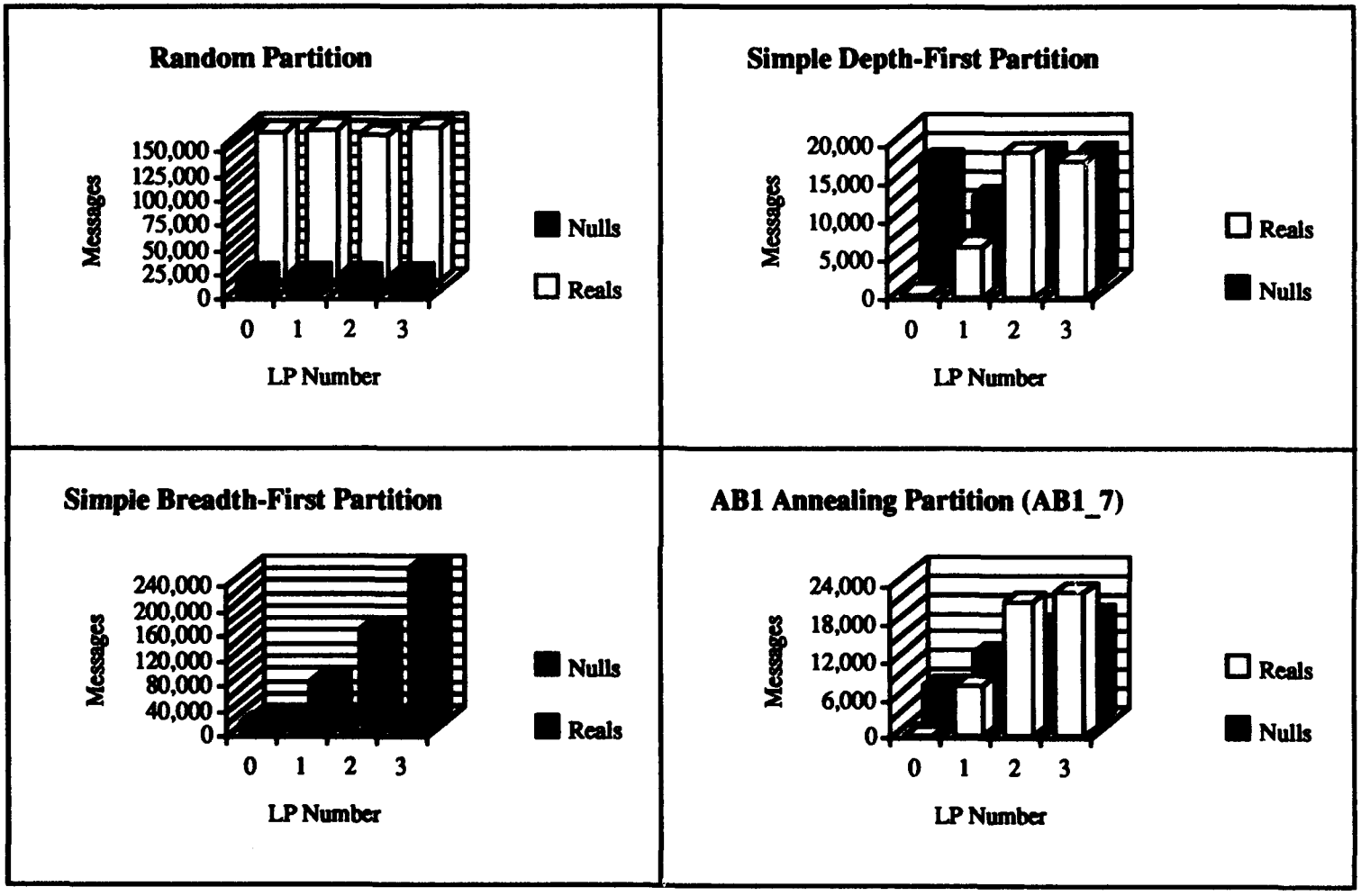


Figure 77. Associative Memory 4 LP Reals Sent vs. Nulls Sent Message Analysis

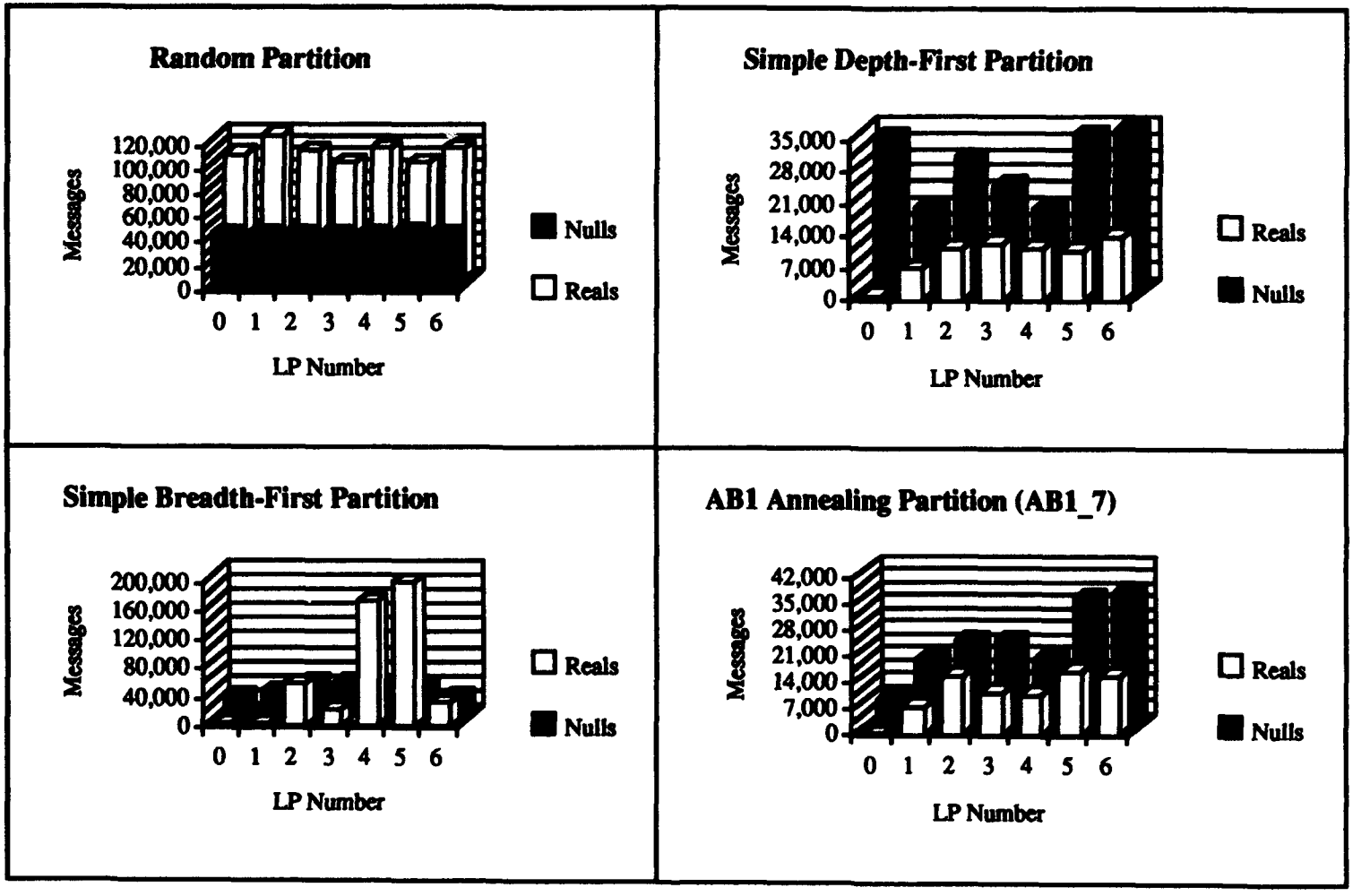


Figure 78. Associative Memory 7 LP Reals Sent vs. Nulls Sent Message Analysis

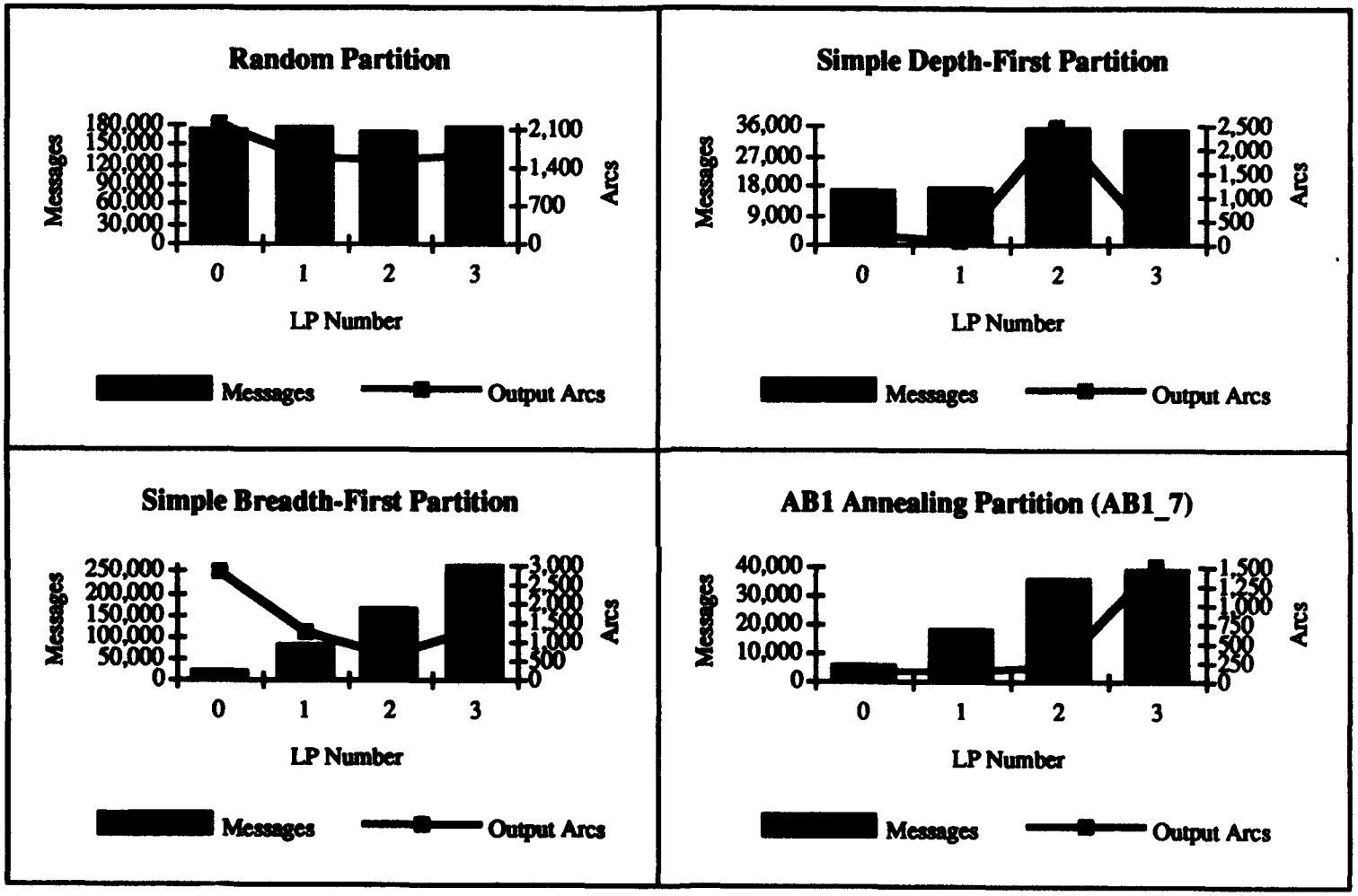


Figure 79. Wallace Tree 4 LP Total Messages Sent vs. Output Arcs

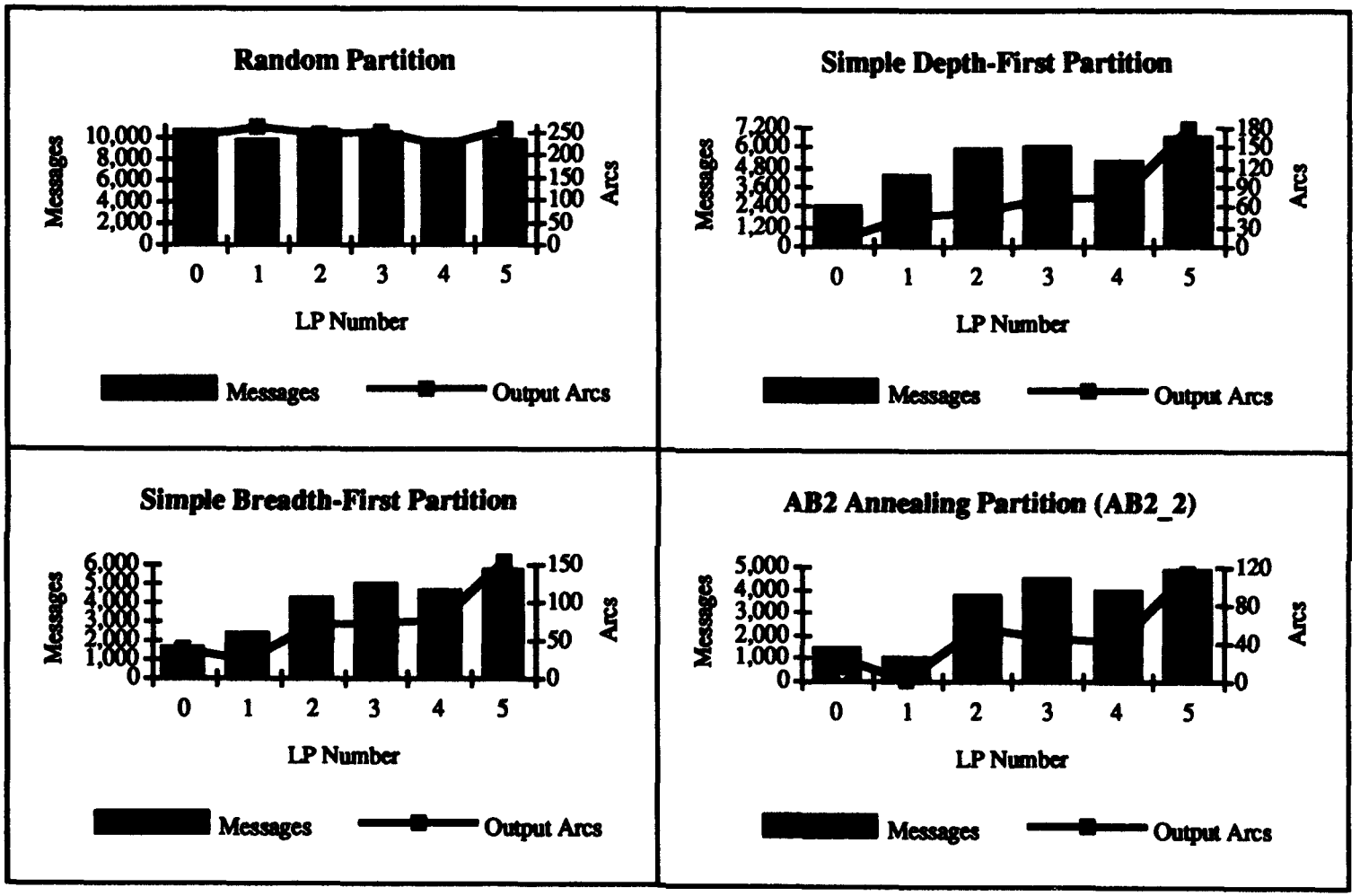


Figure 80. Wallace Tree 6 LP Total Messages Sent vs. Output Arcs

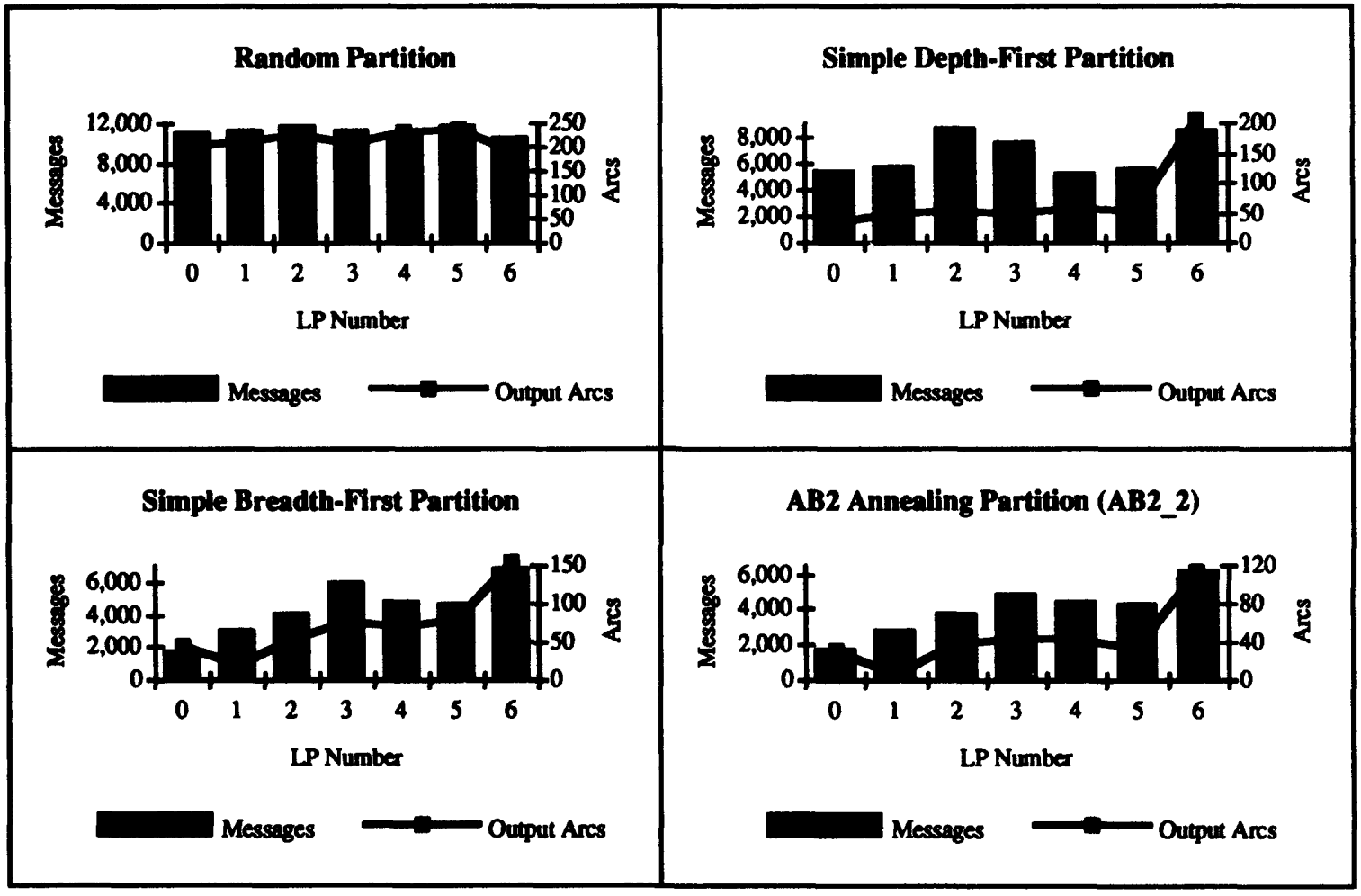


Figure 81. Wallace Tree 7 LP Total Messages Sent vs. Output Arcs

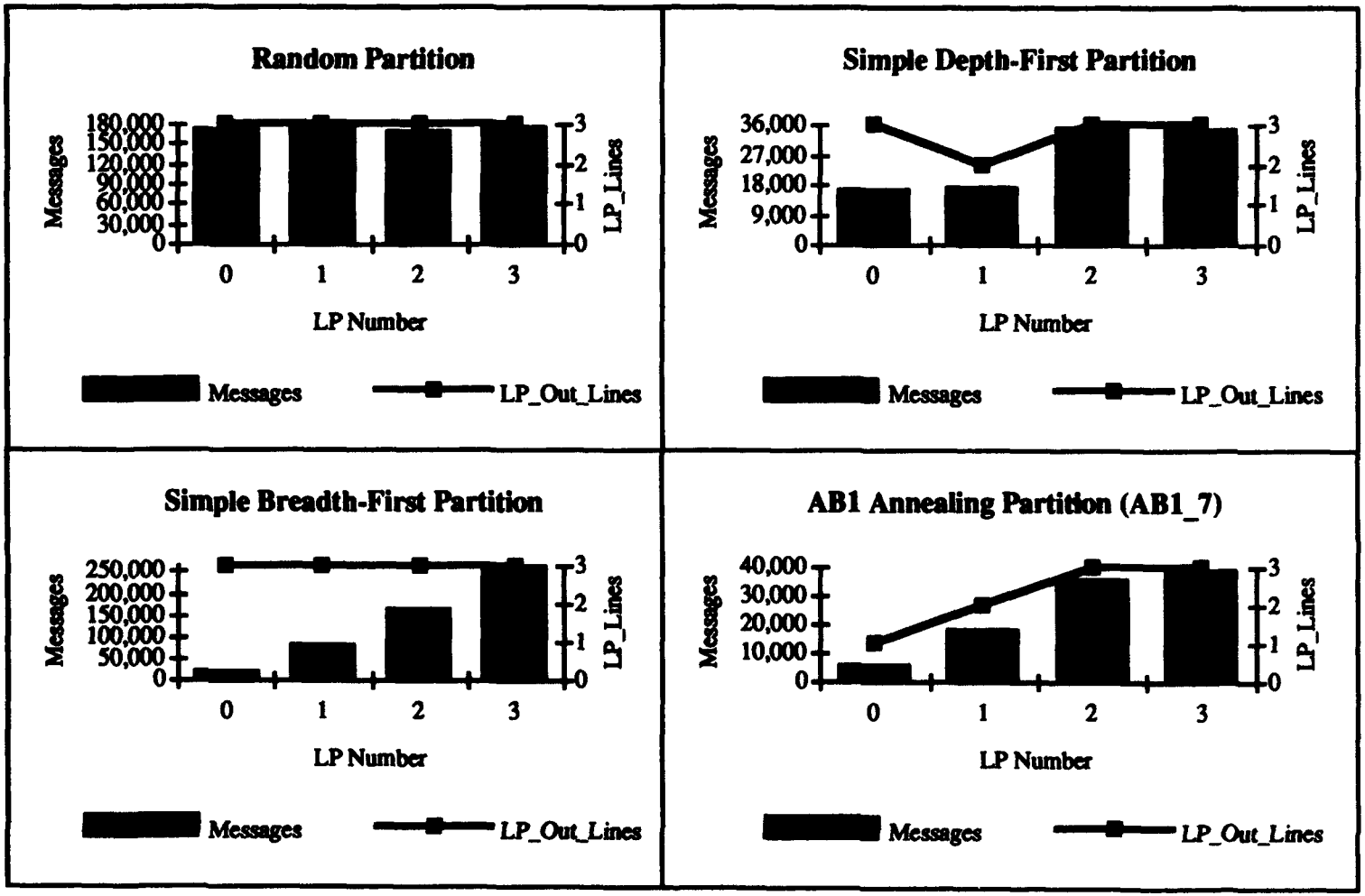


Figure 82. Associative Memory 4 LP Total Messages Sent vs. Output Arcs

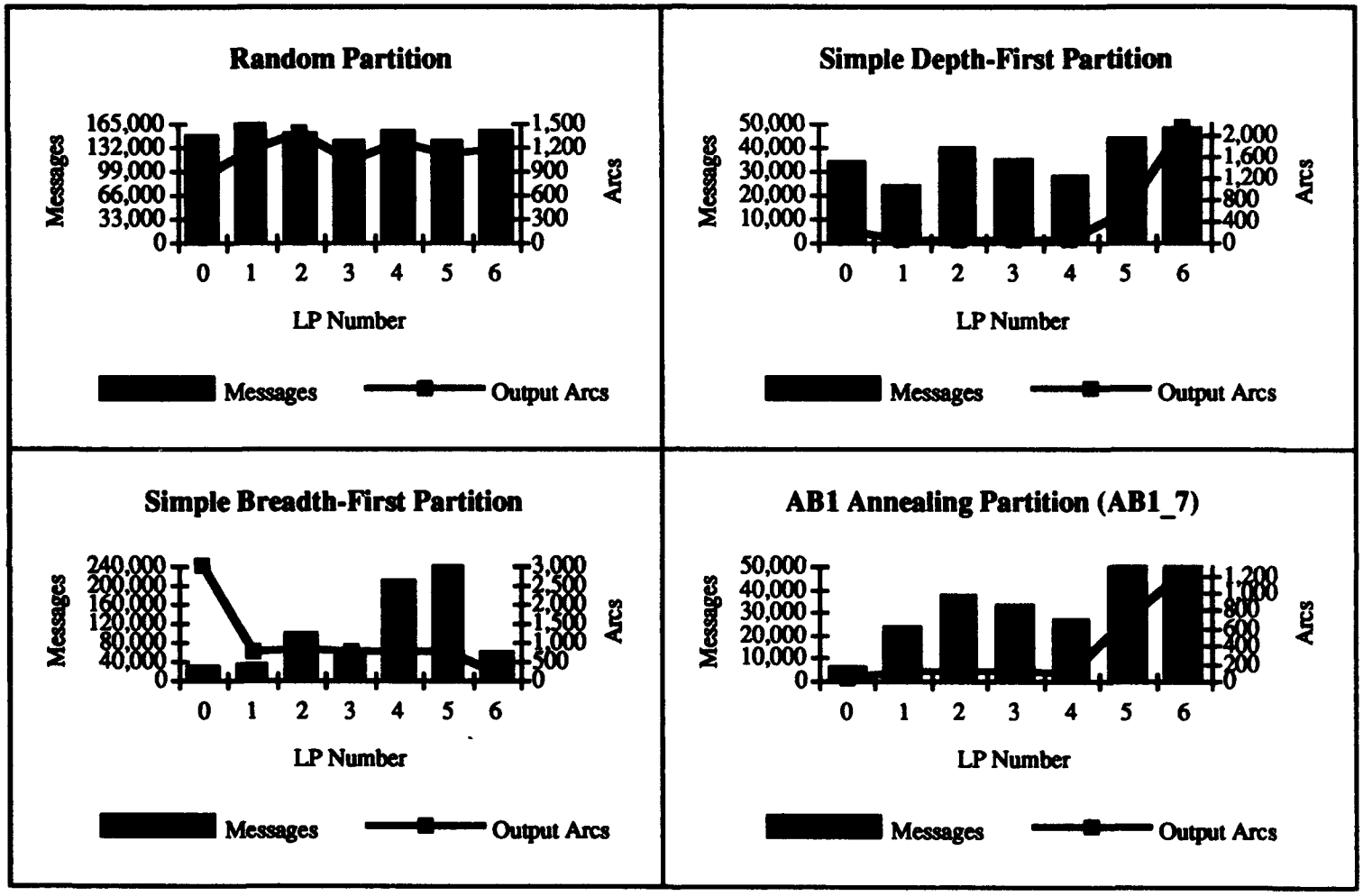


Figure 83. Associative Memory 7 LP Total Messages Sent vs. Output Arcs

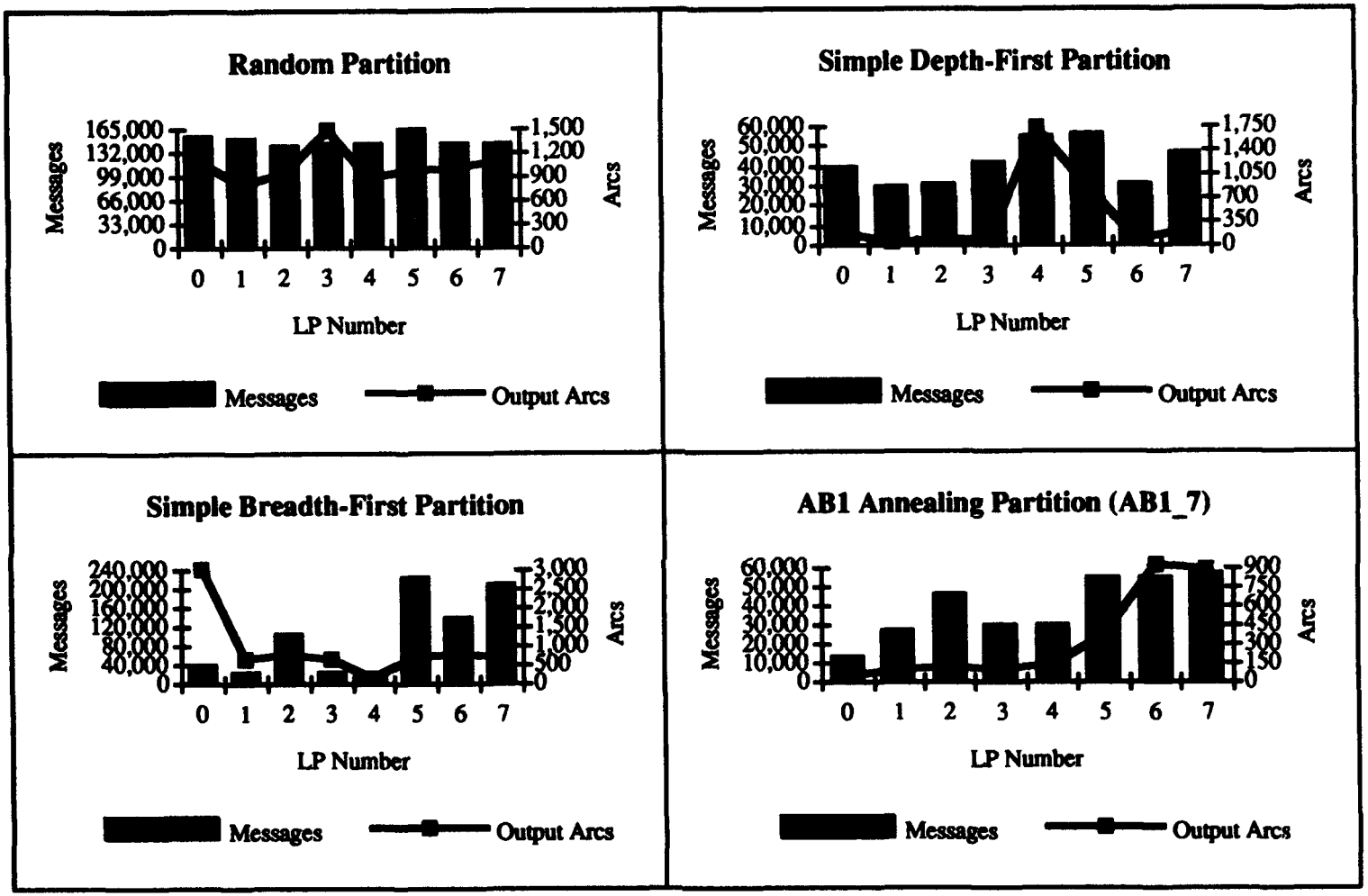


Figure 84. Associative Memory 8 LP Total Messages Sent vs. Output Arcs

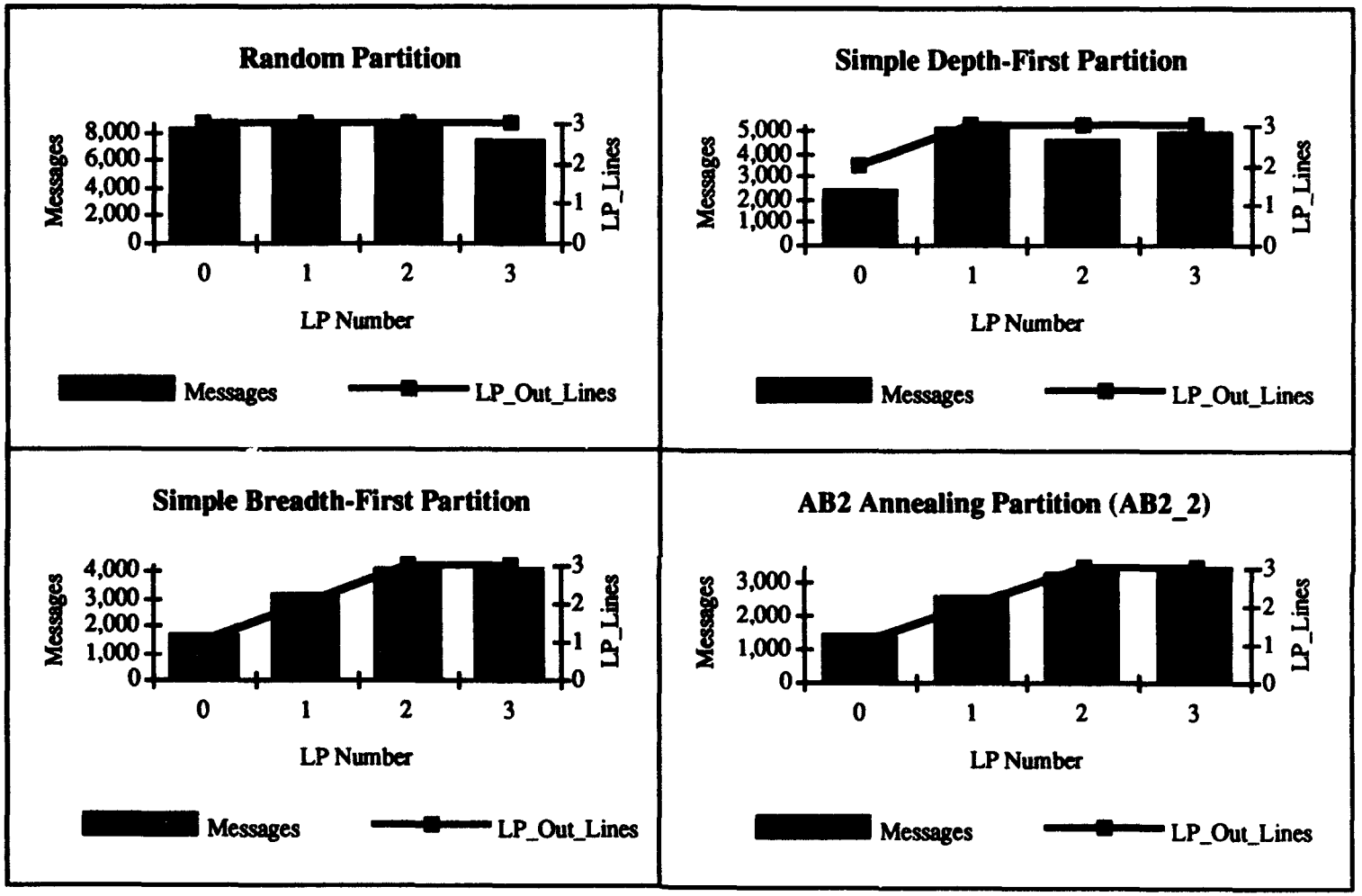


Figure 85. Wallace Tree 4 LP Total Messages Sent vs. LP Output Lines

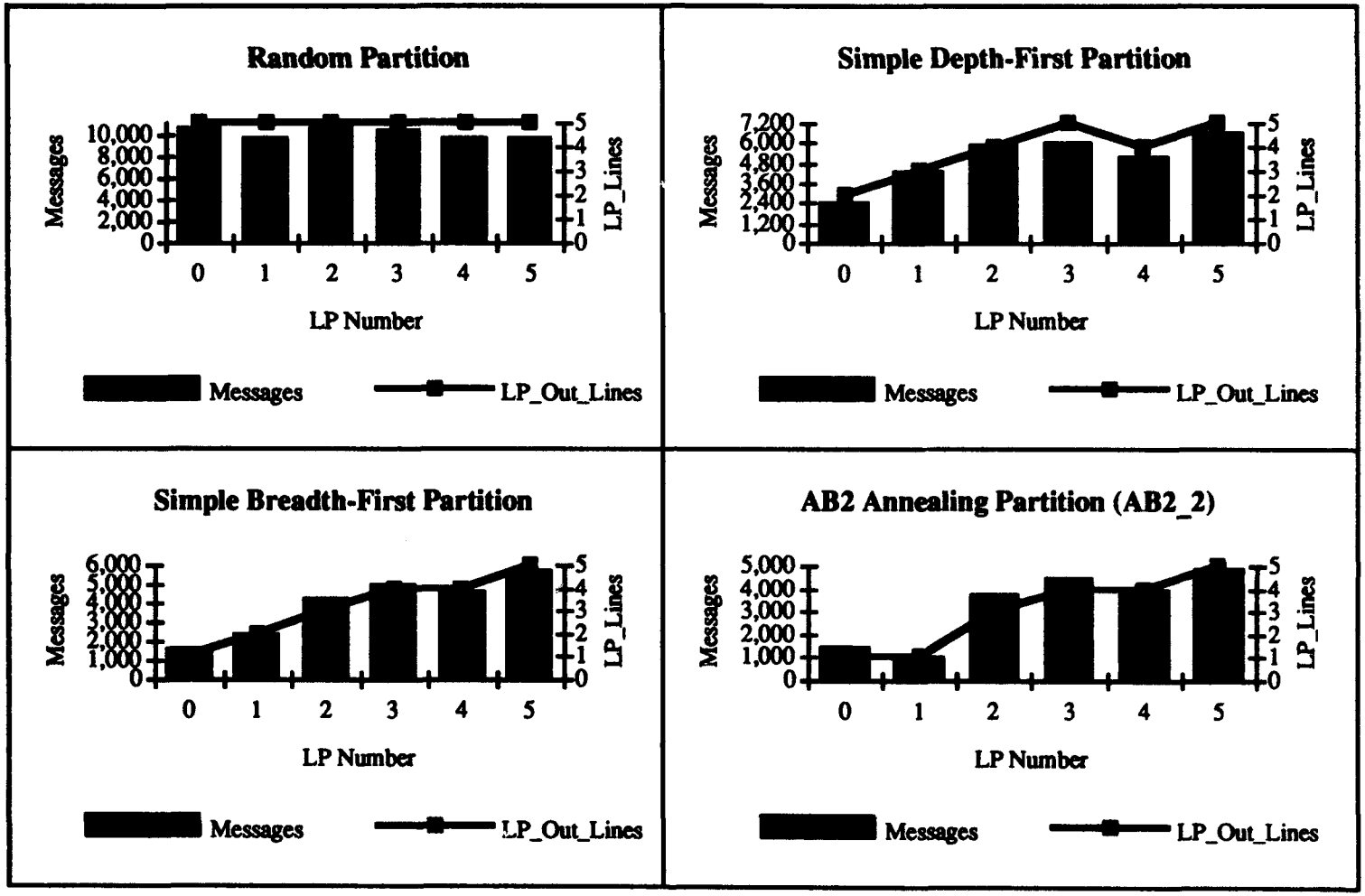


Figure 86. Wallace Tree 6 LP Total Messages Sent vs. LP Output Lines

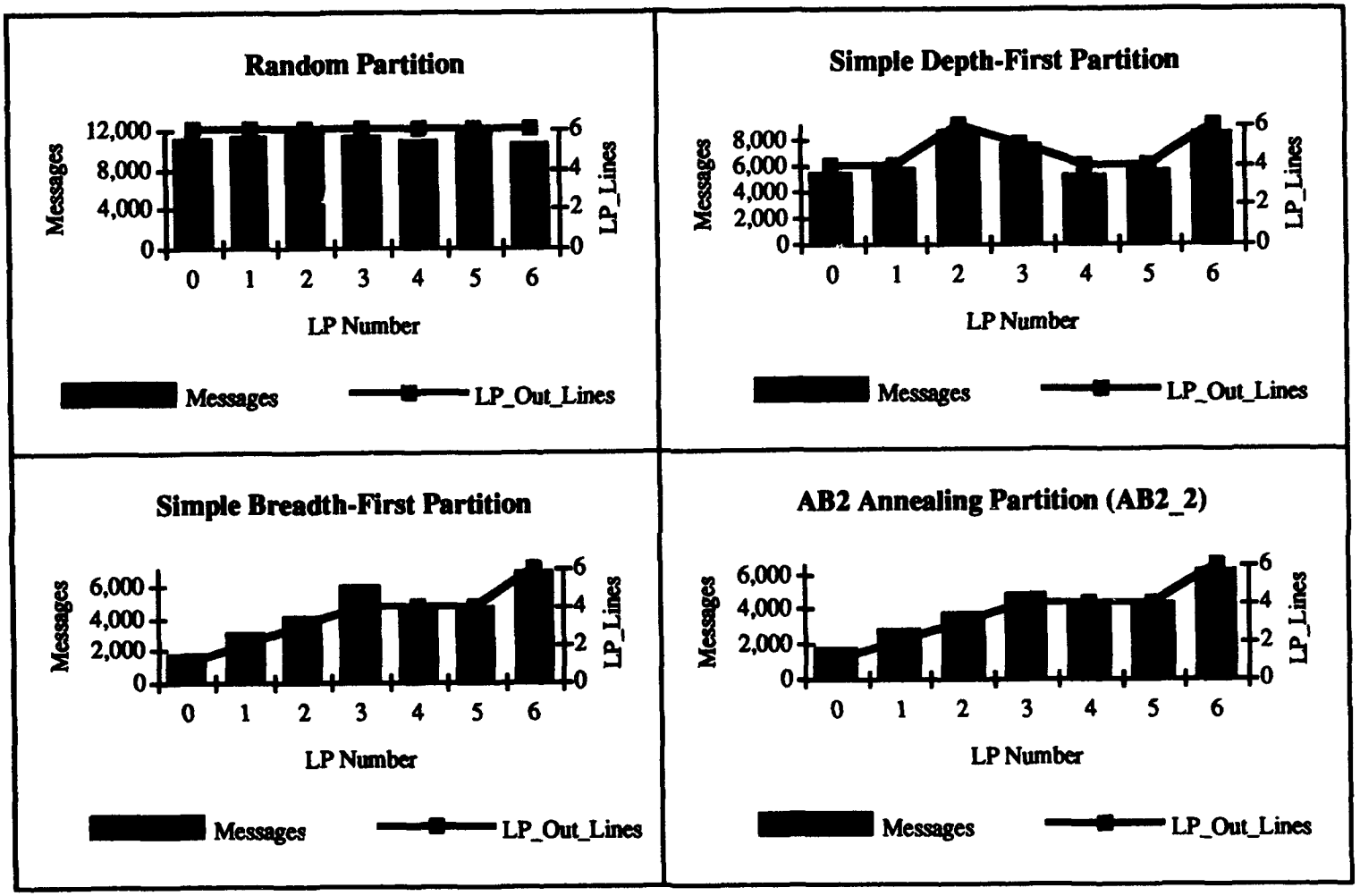


Figure 87. Wallace Tree 7 LP Total Messages Sent vs. LP Output Lines

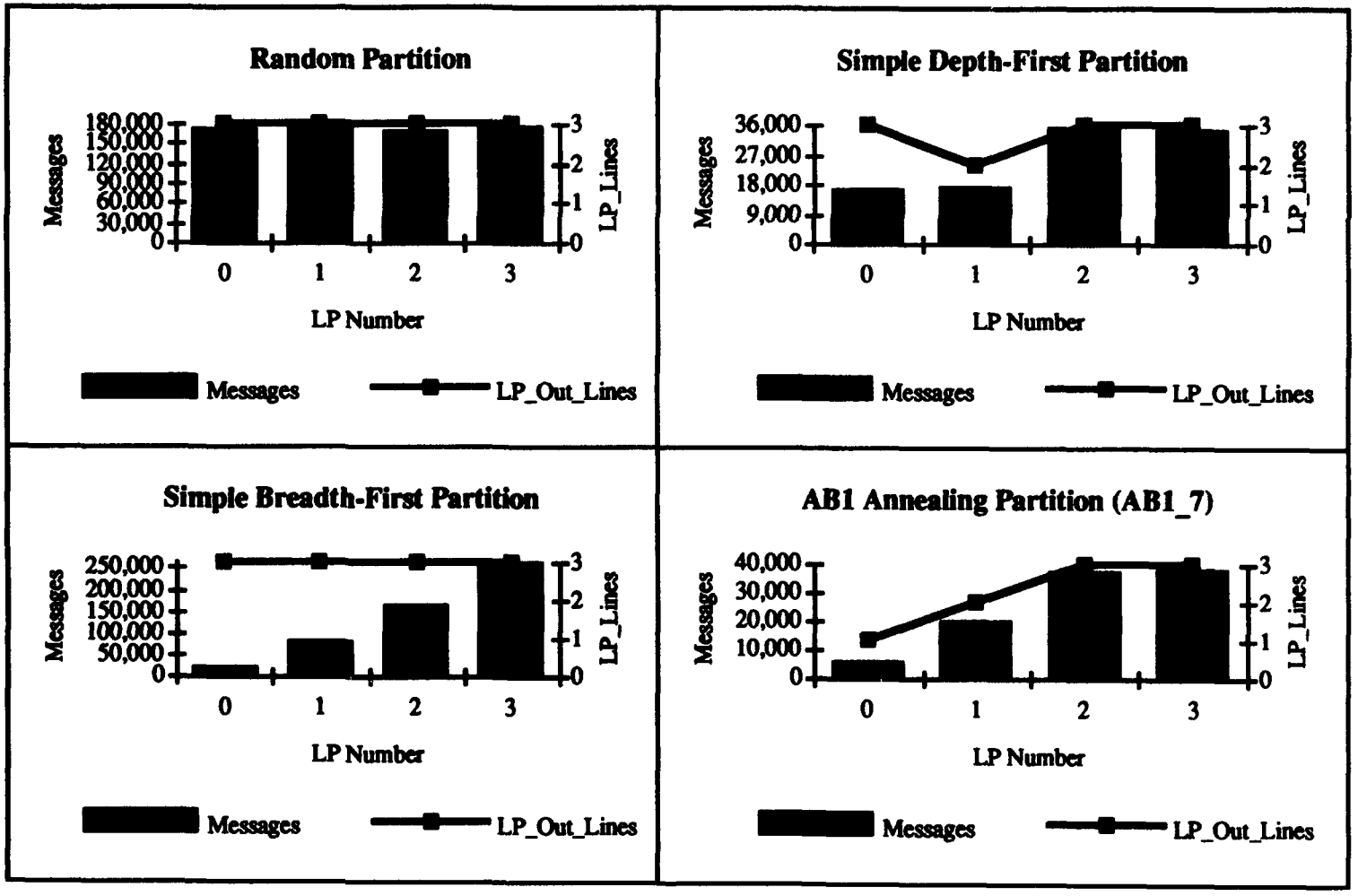


Figure 88. Associative Memory 4 LP Total Messages Sent vs. LP Output Lines

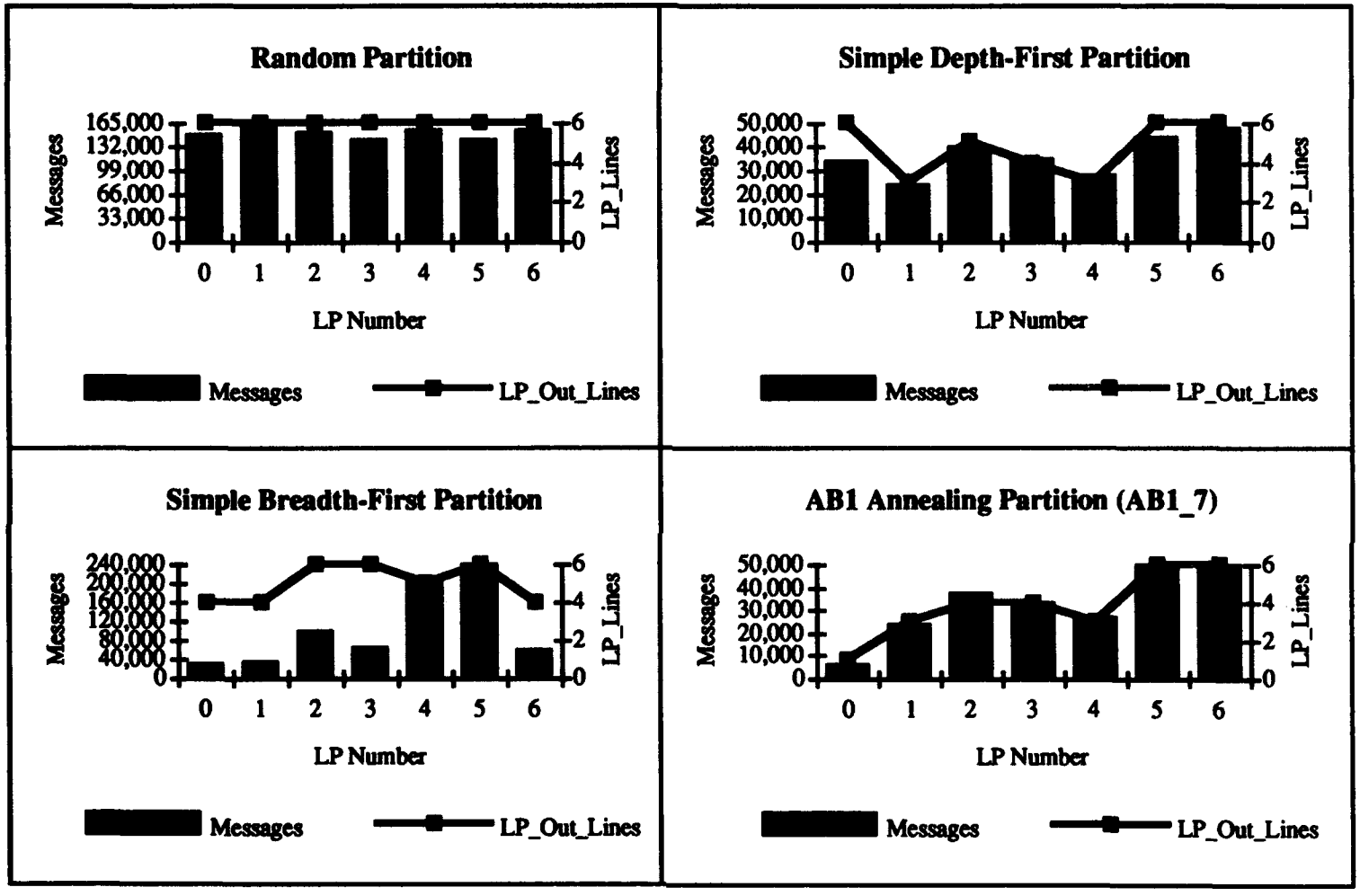


Figure 89. Associative Memory 7 LP Total Messages Sent vs. LP Output Lines

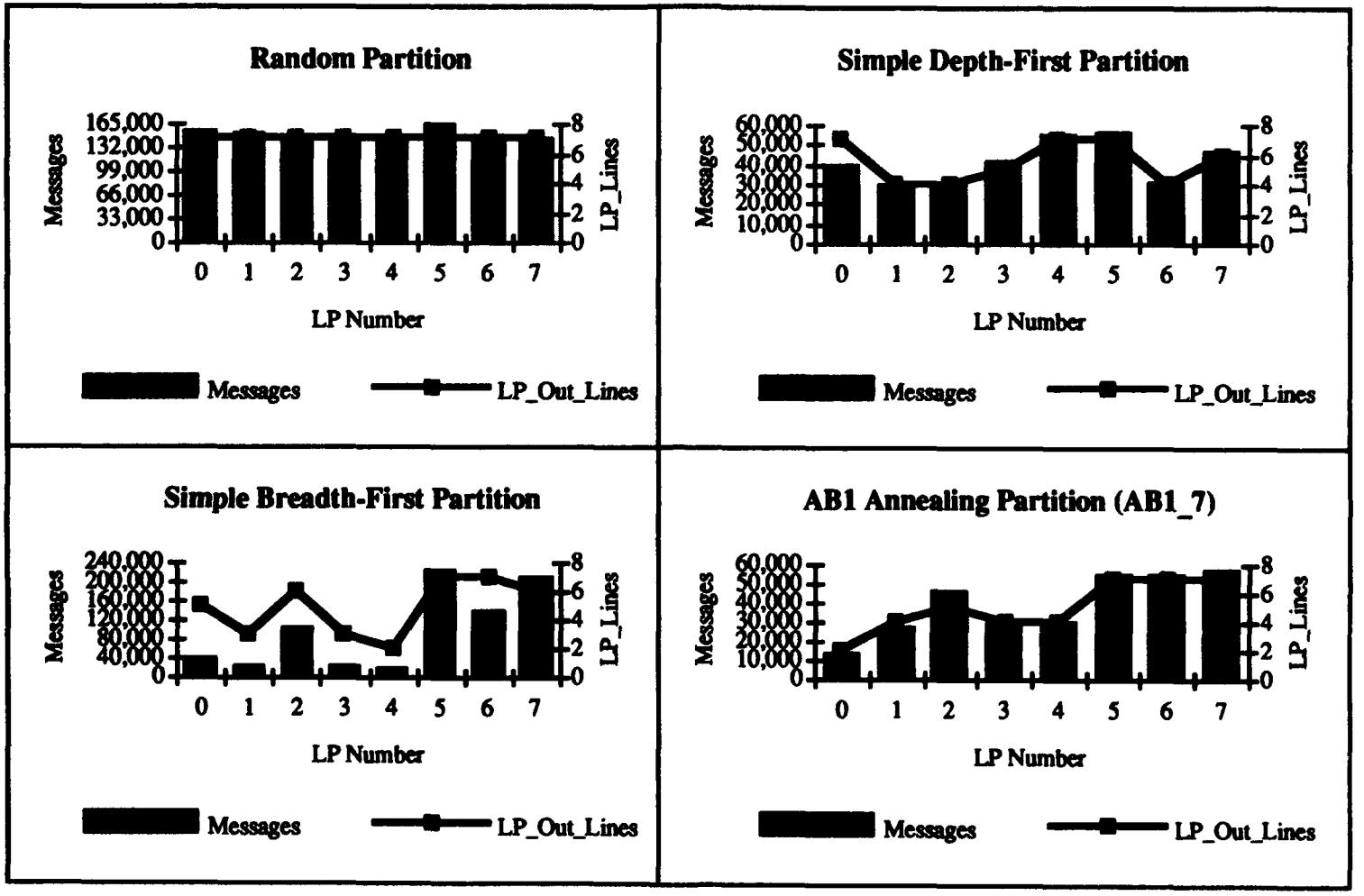


Figure 90. Associative Memory 8 LP Total Messages Sent vs. LP Output Lines

Bibliography

1. Banks, Jerry and John S. Carlson, II. *Discrete-Event System Simulation*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1984.

2. Berman, Francine and Lawrence Snyder. "On Mapping Parallel Algorithms into Parallel Architectures," *Proceedings of the 1984 International Conference on Parallel Processing*. 307–309. 1984.

3. Berman, Francine and Lawrence Snyder. "On Mapping Parallel Algorithms into Parallel Architectures," *Journal of Parallel and Distributed Computing, 4*(5): 439–458 (October 1987).

4. Breeden, Thomas A. *Parallel Simulation of Structural VHDL circuits on Intel Hypercubes*. MS thesis, AFIT/GCE/ENG/92D-01, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1992.

5. Bultan, Tevfik and Cevdet Aykanat. "A New Mapping Heuristic Based on Mean Field Annealing," *Journal of Parallel and Distributed Computing, 16:* 292–305 (December 1992).

6. Carter, Harold, et al. "Semiannual Status Report on the QUEST Project," 15 April 1993.

7. Chandy, K.M. and J. Misra. "Asynchronous Distributed Simulation via a Sequence of Parallel Computations," *Communications of the ACM, 24*(11): 198–206 (April 1981).

8. Comeau, Ronald C. *Parallel Implementation of VHDL Simulations on the Intel iPSC/2 Hypercube*. MS thesis, AFIT/GCS/ENG/91D-03, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1991.

9. Conrad, James M. and Dharma P. Agrawal. "A Graph Partitioning Based Load Balancing Strategy for a Distributed Memory Machine," *Proceedings of the 1992 International Conference on Parallel Processing*. 74–81. 1992.

10. Cormen, Thomas H., Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. Cambridge, MA: McGraw-Hill Book Company, 1992.

11. Fujimoto, Richard M. "Parallel Discrete Event Simulation," *Proceedings of the 1989 Winter Simulation Conference*. 1–34. 1989.

12. Hartrum, Thomas C. "AFIT Guide to SPECTRUM," 22 October 1992. User's Guide.

13. Hennessy, John L. and David A. Patterson. *Computer Architecture: a Quantitative Approach*. San Mateo CA: Morgan Kaufmann Publishers, Inc., 1990.

14. Kernighan, B.W., and S. Lin "An Efficient Heuristic Procedure for Partitioning Graphs," *The Bell System Technical Journal, 49*(1): 291–307 (1970).

15. Roger Lipsett, et al. *VHDL: Hardware Description and Design*. Norwell MA: Kluwer Academic Publishers, 1989.

16. Lo, Virginia M. "Algorithms for Static Task Assignment And Symmetric Contraction in Distributed Computing Systems," *Proceedings of the 1988 International Conference on Parallel Processing*. 239–244. 1988.

17. Manoharan, Sathiamoorthy and Peter Thanisch. "Assigning Dependency Graphs Onto Processor Networks," *Parallel Computing, 17:* 63–73 (1991).

18. Neelamkavil, Francis. *Computer Simulation and Modelling*. Dublin, Ireland: John Wiley & Sons, 1987.

19. Proicou, Michael Chris. *A Distributed Kernel for Simulation of the VHSIC Hardware Description Language*. MS thesis, AFIT/GCS/ENG/89D-14, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1989.

20. Reynolds, Jr., Paul F. and P.M. Dickens. "SPECTRUM: A Parallel Simulation Testbed," *Proceedings of the 4th Annual Hypercube Conference*. 1989.

21. Sadayappan, Ponnuswamy and Fikret Ercal. "Nearest-Neighbor Mapping of Finite Element Graphs onto Processor Meshes," *IEEE Transactions on Computers, C-36*(12): 1408–1424 (December 1987).

22. Schwan, Karsten, et. al. "Mapping Parallel Applications to a Hypercube," *Proceedings of the Second Conference on Hypercube Multiprocessors*. 141–151. 1987.

23. Sporrer, Christian, and Herbert Bauer. "Corolla Partitioning for Distributed Logic Simulation of VLSI-Circuits," *7th Workshop on Parallel and Distributed Simulation (PADS93)*. 85–92. 1993.

24. Symantec Corporation.*Think C Object-Oriented Programming Manual*. 10201 Torre Avenue Cupertino CA 95014, 1991.

25. VHDL Graph Searching Program. Original Version, Sun Ada. Computer Software Source Code. Eric R. Christensen, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, September 1992.

26. Wilson, Gregory V. "A Glossary of Parallel Computing Terminology," *IEEE Parallel & Distributed* Technology, 52–67 (February 1993).

Vita

Captain Kevin L. Kapp was born June 8, 1966, in Louisville, Kentucky. He grew up in Evansville, Indiana, and graduated from William Henry Harrison High School in 1984. He joined the Air Force Reserve Officer Training Corps (AFROTC) program on scholarship at the University of South Florida (USF) in Tampa in the fall of 1984. In April 1989, he received a Bachelor of Science degree in Electrical Engineering, with honors, from USF and was commissioned a Second Lieutenant in the US Air Force. He entered active duty on 15 July 1989 and was assigned to the Electronic Systems Division (now ESC), Hanscom AFB, MA. From July 89 to his entry into AFIT in May 92, he served as a systems engineer and a software test engineer on the Command and Control Information Processing System (C2 IPS) program to upgrade the command and control capabilities of Air Mobility Command.

Permanent Address: 410 S. Hebron Avenue
Evansville, IN 47715

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1993	Master's Thesis

4. TITLE AND SUBTITLE
Partitioning Structural VHDL Circuits for Parallel Execution on Hypercubes

5. FUNDING NUMBERS

6. AUTHOR(S)
Kevin L. Kapp, Capt, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GCE/ENG/93D-07

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
LtCol John Toole
DARPA/CSTO
3701 N. Fairfax Dr.
Arlington, VA 22203

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

Distributing simulations among multiple processors is one approach to reducing VHDL simulation time for large VLSI circuit designs. However, parallel simulation introduces the problem of how to partition the logic gates and system behaviors among the available processors in order to obtain maximum speedup. This research investigates deliberate partitioning algorithms that account for the complex inter-dependency structure of the circuit behaviors. Once an initial partition has been obtained, a border annealing algorithm is used to iteratively improve the partition. In addition, methods of measuring the cost of a partition and relating it to the resulting simulation performance are investigated. Structural circuits ranging from one thousand to over four thousand behaviors are simulated. The deliberate partitions consistently provided superior speedup to a random distribution of the circuit behaviors.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Parallel Simulation, Discrete Event Simulation, Distributed Simulation, Mapping Problem, Static Task Partitioning	211
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL